MASTER COPY KEEP THIS COPY FOR REPRODUCTION PURPOSES

AD-A281 661

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE May 1994

3. REPORT TYPE AND DATES COVERED Final 25 Sep 92-31 Dec 93

4. TITLE AND SUBTITLE
The Highways and Byways of Teaching ADA: Our Backyard Approach

5. FUNDING NUMBERS
DAAL03-92-G-0414

6. AUTHOR(S)
Edward Calusinski
Tzilla Elrad
Thomas Grace

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Illinois Institute of Technology
Chicago, IL 60616

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
U.S. Army Research Office
P.O. Box 12211
Research Triangle Park, NC 27709-2211

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
ARO 30997.1-MA

11. SUPPLEMENTARY NOTES
The views, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Ada is a modern language, for modern students that solve modern problems. Ada was designed to be a language that promotes the goals of modern software engineering. It promotes modifiability, efficiency, reliability, and understandability. Ada was also designed to support the principles of modern software engineering. It promotes data abstraction, information hiding, modularity, localization, uniformity, completeness and confirmability. Ada's original design chose program readability over ease of writing. This attribute promotes code understandability, which

(continued on reverse side)

14. SUBJECT TERMS
ADA, Computer Science, Language, ADA Language, Software Engineering

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev 2-89)
Prescribed by ANSI Std Z39-18
298-102

94 7 11 206

helps prevent erroneous and error-prone programs. The Ada language supports separate compilation units. This helps in program development and maintenance, which is helpful when developing large, complex software engineering projects. Given all the underlying features of Ada, it is apparent that Ada is an excellent language to use when teaching students the principles of computer science.

THE HIGHWAYS AND BYWAYS OF TEACHING ADA: OUR BACKYARD APPROACH

FINAL REPORT

EDWARD CALUSINSKI
DR. TZILLA ELRAD
DR. THOMAS GRACE

Accesion For		
NTIS CRA&I		X
DTIC TAB		☐
Unannounced		☐
Justification		
By		
Dist ibution /		
Availability Codes		
Dist	Avail and / or Special	
A-1		

APRIL 15, 1994

U.S. ARMY RESEARCH OFFICE

30997-MA

DAAL03-92-G-0414

ILLINOIS INSTITUTE OF TECHNOLOGY

THE VIEWS, OPINIONS, AND/OR FINDINGS CONTAINED IN THIS REPORT ARE THOSE OF THE AUTHOR(S) AND SHOULD NOT BE CONSTRUED AS AN OFFICIAL DEPARTMENT OF THE ARMY POSITION, POLICY, OR DECISION, UNLESS SO DESIGNATED BY OTHER DOCUMENTATION.

Table of Contents

The Highways and Byways of Teaching Ada: Our Backyard Approach

BACKGROUND

The Illinois Institute of Technology is a private, medium-sized, coeducational university, which offers undergraduate and graduate programs through the Institute of Design and six schools and colleges: the College of Engineering and Sciences; the College of Liberal Arts; the College of Architecture; the School of Business; the College of Law and the Graduate School. The 120-acre main campus is located about three miles south of Chicago's Loop. The Computer Science Department is part of the College of Engineering and Sciences which offers Bachelor of Science, Bachelor of Arts, Masters of Science, Masters of Science for Teachers, and Doctor of Philosophy degrees. The curriculum is software centered, which is attractive to all types of majors. Our platform of choice is the IBM PC. This makes our program more interesting to students who have majors other than computer science. This accounts for our large number of students, with majors other than computer science, who enroll in our courses. IIT has an unique situation in that we offer students an education through IITV (Interactive Instructional Television Network). IITV is a live, talk-back television system that enables participating companies to offer IIT's educational programs to their employees at their places of business. The network links classroom-studios on campus with receiving classrooms at industrial and business locations so that employee-students can join IIT day and evening classes without traveling to campus.

WHY USE ADA

Ada is a modern language, for modern students that solve modern problems. Ada was designed to be a language that promotes the goals of modern software engineering. It promotes

modifiability, efficiency, reliability, and understandability. Ada was also designed to support the principles of modern software engineering. It promotes data abstraction, information hiding, modularity, localization, uniformity, completeness and confirmability. Ada's original design chose program readability over ease of writing. This attribute promotes code understandability, which helps prevent erroneous and error-prone programs. The Ada language supports separate compilation units. This helps in program development and maintenance, which is helpful when developing large, complex software engineering projects. Given all the underlying features of Ada, it is apparent that Ada is an excellent language to use when teaching students the principles of computer science.

OUR APPROACH TO INTEGRATING ADA

When we were first contemplating integrating Ada into our curriculum, we asked ourselves the question, "Why is Ada not already integrated?". We came to the conclusion that one, the students and faculty had very little to no exposure or awareness of Ada. Two, the DoD and DoD related industries are very limited in our area. Even though Ada is not just limited to the DoD, our students would have little interest in learning Ada, if they could not apply it. Three, Ada is very rich in its syntax, but our curriculum in the past has favored simpler languages like C, Pascal, Lisp/Scheme which are thought to be "easier to learn". Four; A large population of our international students end up returning to there native countries. Most of these countries have little or no use for Ada, which makes learning Ada unattractive. We consider this a "backyard approach". There is no real reason for us to teach Ada, except we realize the capabilities and powerful features that Ada possesses. Because we are not in the "mainstream" Ada community,

the acceptance of this becomes difficult. We are presented with the questions "Where do we integrate Ada into are curriculum?" and "How can this be done given our current curriculum?".

WHERE AND HOW TO INTEGRATE ADA

At first we thought of integrating Ada in our entry level courses. This seems to work well for other Universities, but at IIT we felt that to incorporate the more advanced and best features that Ada has to offer, we would need to integrate Ada at higher level. We based this on the fact that most first year students are unfamiliar with the rigors of software development, and would therefor under utilize its features and richness. Our curriculum is developed in small increments due to the accrediting concerns, so a major change would set us back. Our second choice was to develop a new sequence of software engineering courses around Ada. This was a better idea, but our undergraduate curriculum is full and adding a new sequence might not attract the interest we would need to maintain the courses. This idea also moves away from our objective of integrating Ada. This idea would isolate Ada rather than integrating it. We finally came to the conclusion that the best way for IIT to integrate Ada was to put it into our existing, and popular (i.e. required) mainstream upper-division courses. We based this decision with the following things in mind. This implementation does not require a "special audience". Our students, once at the junior or senior level, will be able to fully appreciate the richness of Ada. Also, there is a good probability that Ada would migrate into more of our curriculum.

ACCOMPLISHMENTS

Identification and Acquisition of Resources.

- Substantial resources via FTP on Internet (code, documentation, etc.).
- Ada Tutor -- shareware; copies are available and distributed to students.
- LearnAda, by AETech, which is a tutoring system and Ada compiler available to students on the IIT PCs

- A comprehensive manual "Getting Started With Ada" for the AETech software which includes the ANSI/MIL-STD-1815A Ada Reference Manual.
- Additional Ada literature developed by the Teaching Assistants which has helped support course development and Ada integration.
- We have hired and additional part-time instructor to teach Ada.
- A 16 week Ada course available on video tape (VHS format).

COURSE DESCRIPTION OF CLASSES THAT INCORPORATE ADA

CS440: Programming Languages and Translators.

This course is a general introduction to theory and structure of languages. It covers several programming paradigms (imperative, declarative, object-oriented, functional, etc.) There has been a substantial amount of material added regarding Ada, especially:

rich syntax
generics
types and type checking
overloading
proposed Ada9X features

The integration is included in the lecture material, homework and class exercises, examination topics and questions.

CS450: Operating Systems.

This a standard undergraduate course on operating systems. Its topics include CPU scheduling, process management, memory management, file systems, multitasking, concurrency, synchronization, security, etc. Ada is now the base language for this course. This is, of course, especially important because of the Ada tasking model. Materials developed include examples in

lecture, exercises, and examination materials. Students are strongly encouraged to submit programming assignments in Ada (although, for symmetry with our other courses, a choice of languages is provided).

CS495: Safety Critical Software Engineering With Ada.

This course provides an in-depth examination of the principles behind development of software intended for use in critical situations. Such situations include air-traffic control, medical applications, defense weaponry, space exploration, etc. This course also emphasizes testing, maintenance and reusability of code. Ada is the only language used in this course. Extensive programming experience in Ada, emphasizing more advanced features, is provided.

CS545: Concurrent Programming.

This is a graduate course, but it is available to undergraduates with an advisor's approval. Intensive examination and comparison of the various language testing models and there richness of support for concurrency, especially in real-time systems. Ada is the base language for this course, and extensive programming experience in Ada is provided. Ada9X issues are also explored including protected types, asynchronous transfer of control, mutual control, race conditions, etc.

CONCLUSION

We believe that our approach is fairly novel. We have not tried to duplicate efforts of other educators to develop CS1 or CS2 courses based on Ada. We have not tried to develop and "Ada Track" of courses. Instead, we are integrating Ada into the very core of our undergraduate curriculum, where Ada can be used to its fullest potential. By implementing this "backyard" approach to teaching Ada, we have been able to break the trend of other successful institutions by implementing Ada at a higher level. We have been able to take advantage of all the features that Ada has to offer, from the most basic to the most advanced. By doing this, we are able to teach modern day concepts, using a modern day language, in the comfort of our own backyard.

APPENDIX A

CS450 OPERATING SYSTEMS COURSE

SYLLABUS AND ADA LECTURE NOTES

Lecture One

Textbook

Abraham Silberschatz, James L. Peterson, Peter B. Galvin
"Operating System Concepts", Addison-Wesley, third edition, 1991, Part 1, Chapters 1, 2 and 3

References

- Andrew S. Tanenbaum
  "Modern Operating Systems", Prentice-Hall, 1992, Part 1, Chapter 1
- Jean Bacon
  "Concurrent Systems - An Integrated Approach to Operating Systems, Database, and Distributed Systems", Addison-Wesley, 1993, Chapter 1 (1.1.1, 1.1.3) and Chapter 3 (3.2 - 3.5)

Goals

- To present organization of the course
- To review history and evolution of the computer and operating systems
- To explain the notion of the operating system
- To discuss computer system structures
- To discuss operating system structures

Content

- Content of the course
- History, Evolution, and Philosophy of Operating Systems (3 hours)

Lecture One

Organization of the Course

- Course Syllabus (example in the appendix)
- Overview of the Course Material
- Overview of the First Lecture

History and Evolution of Operating Systems

- First generation (1945 - 1955)
  Vacuum Tube and Plugboards
  First true digital computer - Charles Babbage (1792 - 1871) (unsuccessful)
  Howard Aiken (Harvard University), John von Neumann (Princeton University), J. Presper Eckert and William Mauchley (University of Pennsylvania), Konrad Zuse (Germany), all succeeded in building calculating machines using vacuum tubes
    - huge
    - slow
    - machine language
    - wiring up, plugboards to control the machine's basic functions
    - programming languages and assembly language (unknown)
    - operating systems (unheard of)
    - mode of operation: sign up for a block of time
    - introduction of punched cards instead of plugboards (1950)
- Second generation (1955 - 1965)
  Transistors and batch systems
    - off-line operation
    - batch systems
    - control cards - modern JCL and command interpreters
    - special program (the ancestor of today's operating system) (Fortran Monitor System - FMS and IBSYS - IMB 7094)
- Third generation (1965 - 1980)
  Small-scale integrated circuits and multiprogramming
    - general purpose machines
    - huge operating systems (OS/360)
    - *multiprogramming*: several jobs in memory ready for execution
    - time-sharing: variant of multiprogramming (CTSS - MIT, MULTICS - MULTiplexed Information and Computer System)
    - buffering: overlapping the I/O of a job with its own computation (I/O-bound and CPU-bound jobs)
    - spooling (Simultaneous Peripheral Operation On Line): one program might have been executing while I/O occurred for other jobs
- Fourth Generation (1980 - 1990)
  Personal Computers
    - LSI circuits
    - workstations
    - highly interactive computing power with excellent graphics and user-friendly software (MS-DOS,OS/2, UNIX)
    - network and distributed operating systems

Traditional Operating Systems
  In the past most computers ran standalone and most operating systems were designed to run on a single processor
  Centralized systems: single CPU, its memory, peripherals and some terminals
Distributed Operating Systems
  Computers may be networked together, making distributed operating systems more important
  In mid 1980s:
    - development of powerful microprocessors
    - development of high-speed local area networks (LANs)
  Result: large number of CPUs connected by a high-speed network
  Need for radically different software:
    - motivation
    - goals
    - advantages
    - disadvantages
  Modern computers: one or more processors, main memory, clocks, terminals, disks, network interfaces and other I/O devices
  - tightly-coupled systems: processors share memory and a clock
  - loosely-coupled systems: each processor has its own memory
Real-Time Operating Systems
  Processing must be done within the defined constraints

Computer System Structures

- Older systems - data transfer under the CPU control - busy waiting

multiprogram min g
time sharing — benefit from the overlap of CPU and I/O operations ⇒ mechanism needed to allow synchronization of operation ⇒

- interrupt-driven data transfer
    - polling
    - vectored interrupt system

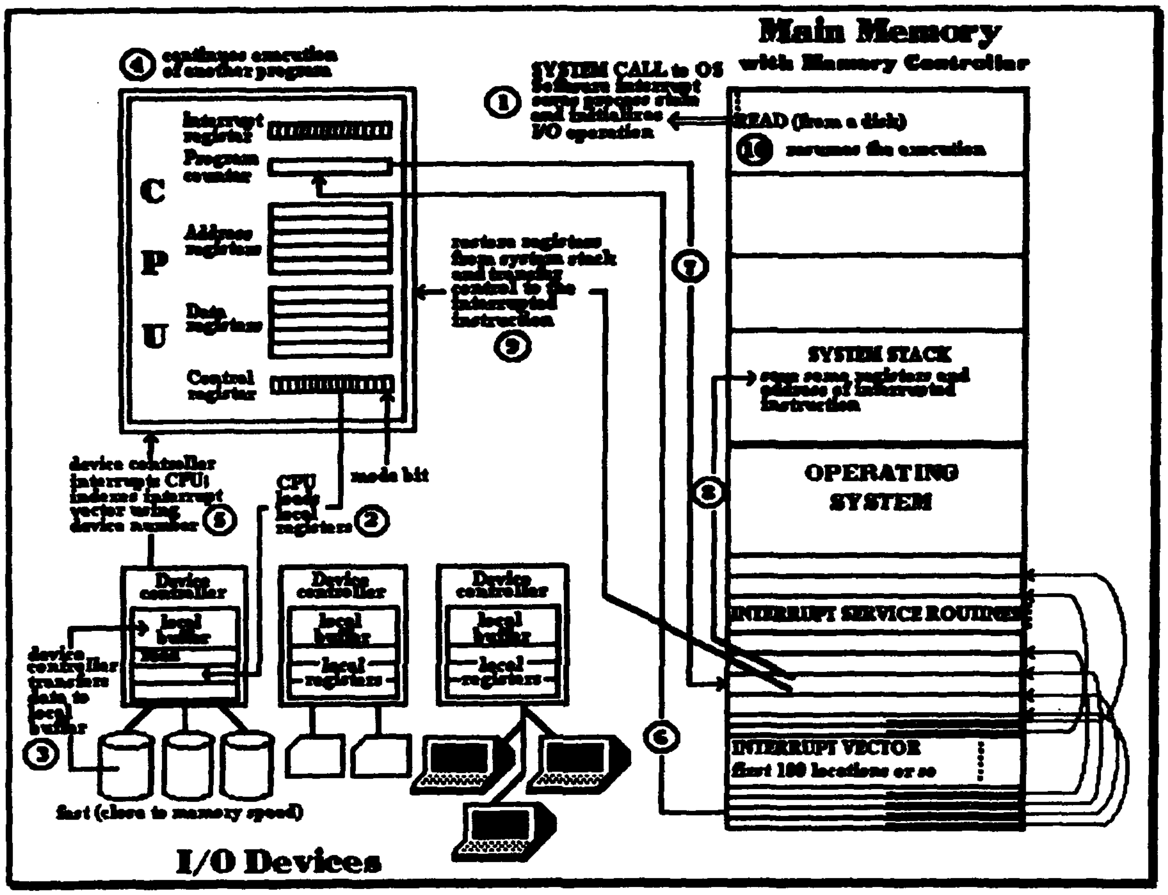


Figure 1.1. Example of the interrupt-driven data transfer

- direct-memory-access (DMA) data transfer

  During the transfer the disk controller is transfering data to or from memory at the same time as the processor is fetching instructions from memory and reading and writing data operands in memory. The memory controller ensures that only one of them at once is making a memory access. The disk controller may access memory between an instruction fetch and a data access which is part of the instruction execution; this is called cycle stealing. DMA slows down the rate at which the processor execute instructions.

  High speed I/O devices

  DMA Controller

  Transfer of an entire block of data
  - to (or from) its own buffer
  - from (or to) memory
  - directly (no intervention by the CPU).

  One interrupt per block

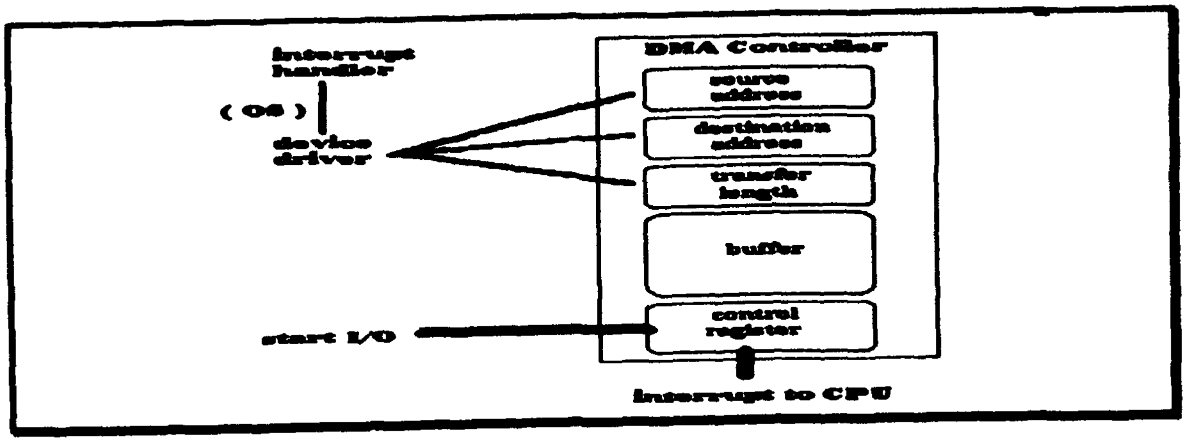
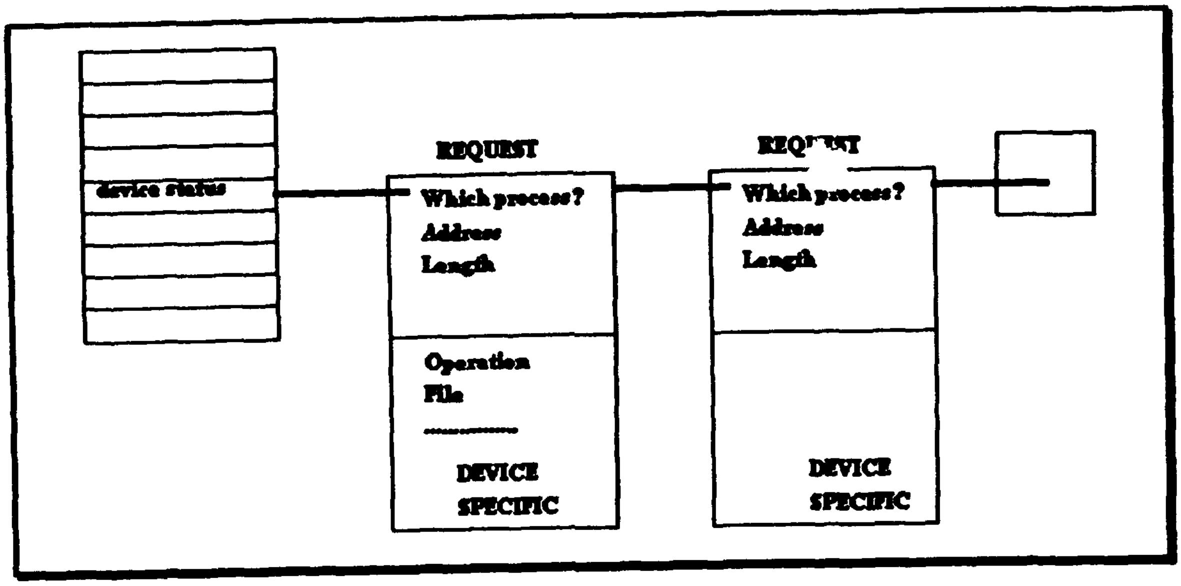


Figure 1.2. DMA controller

Operating System

Find a way to shield programmers from the complexity of the hardware:
To put a layer of software on top of the bare hardware to manage all parts of the system and present the user with an interface or virtual machine that is easier to understand and program.
Keep track of many I/O requests at the same time

Device-Status Table with Request Lists



- System call (program)
  - Put I/O request into the request list (OS)
  - Return control to one of the programs (OS)
- Computation (program)
- Interrupt (device - hardware)
  - Identify the source (OS)
  - Index Device-Status Table (read or modify the entry) (OS)
  - If this is completion
    - Interrupt Service Routine (OS)
    - Start new job from request list (if any) (OS)
    - Return control to one of the programs (OS)
- Computation (user program)

Dual Mode Operation

- Single-user (programmer operated)
- Control transferred to OS
    - especially management of I/O
    - share of system resources ● increased problems
- Increased problems
    - Error ● many jobs could be adversely affected
    - All errors must be detected
        - \- by hardware (illegal instruction ● trap,.....)
- Modes of operation
    - user mode
    - monitor (supervisor, system) mode

Trap or interrupt
    user mode ● monitor mode
After finishing its job OS switches from
    monitor mode ● user mode
Some machine instructions are privileged ● monitor mode

Lecture One

Privileged Instructions

System calls - may be used from high-level language

- all I/O instructions
- change of base and limit registers
- timer operation
  - to prevent infinite loops of never returning control to OS - fixed-rate clock and counter; OS sets the counter ● clock increments it ● zero generates an interrupt
  - time-sharing - O□
  - current time
- halt

Operating-System Services

- OS provides environment for the execution of the programs
- Service may vary
- Load program
- Run program
- Terminate (normally or abnormally)
- I/O operations
- Create files
- Delete files
- Read files
- Write files
- Communication
  - Processes on the same computer
  - Processes on the separated computers
  - By means of shared memory
  - By means of message passing
- Error protection
- Error handling
- Resource allocation
- Accounting
- Protection

System Calls

- Process control
- File manipulation
- Device manipulation
- Information maintenance
- Communications

System Programs

- File manipulation
- Status information
- File modification
- Programming - language support
- Program loading and execution
- Communications
- Applications

Lecture One

Operating System Structures

Huge system ● design requires partitioning into smaller pieces ● carefully designed input, output, functions

- Process management
    - System activities
    - Process - program in execution
        - Program is *passive* entity
        - Process is *active* entity (PC specifies next instruction to be executed)
    - System and user processes
    - Input: CPU time, memory, files, I/O devices
    - Functions: 1. Creation (deletion) of the process
        2. Suspension (resumption) of the process
        3. Synchronization
        4. Communication
        5. Deadlock handling
- Main-memory management
    - Main memory is a large array of words or bytes addressable and quickly accessible
    - Instruction-fetch cycle reads instructions
    - Data-fetch cycle reads or writes data
    - Program [illegible] loaded to absolute addresses into the main memory
    - Memory-management schemes ★ hardware support
    - Functions: 1. Bookkeeping (what is used and by whom)
        2. Who will be loaded into available space
        3. Allocate and deallocate memory space
- Secondary-memory management
    - Functions: 1. Free-space management
        2. Storage allocation
        3. Disk scheduling
- I/O system management
    - Functions: 1. Buffer-caching system
        2. General device driver interface
        3. Drivers for specific hardware devices
- File management
    - Information can be stored in several different physical forms: magnetic disk, magnetic tape,...
    - Logical storage unit (file) [illegible] Physical storage unit
    - Functions: 1. Creation (deletion) of file and directories
        2. Primitives for the manipulation of files and directories
        3. Mapping files onto secondary storage
        4. Backup on stable storage media
- Protection system
    - Memory-addressing hardware
    - Controlling access to resources by means of specification
    - Controlling access to resources by means of enforcement
- Networking
    - Distributed system does not share memory or clock
    - Functions: 1. Routing and connection strategies
        2. Contention and security
- Command-interpreter system
    - Part of the kernel (DOS) or running when job is initiated

Lecture Two

Textbook

Abraham Silberschatz, James L. Peterson, Peter B. Galvin
"Operating System Concepts", Addison-Wesley, third edition, 1991, Part 2, Chapter 4 (4.1, 4.2)

References

- Andrew S. Tanenbaum
  "Modern Operating Systems", Prentice-Hall, 1992, Part 1, Chapter 2 (2.1)
- Jean Bacon
  "Concurrent Systems - An Integrated Approach to Operating Systems, Database, and Distributed Systems", Addison-Wesley, 1993, Chapter 6

Goals

- To present the concept of the process
- To give examples of process partitioning
- To discuss and explain process state model
- To discuss solutions used in actual operating systems

Content

- Tasking and Processes (3 hours)
  - Process Concept
  - Process State Model
  - Implementation of Processes

Lecture Two

Process Concept

Notion of the process (job [batch], user program = task [time-shared])

- unit of work in a system
  - (operating) system processes
  - user processes
- active entity ● program in execution (with a program counter specifying the next instruction to execute)
- program code
  current activity
  stack (subroutine parameters, return addresses, temporary variables,...)
  data section (global variables)

  Defined by the resources it uses and by the location at which it is executing.

Types of processes

- system processes
- user processes
- CPU-bound processes
- I/O-bound processes
- sequential processes
  - execution of a process must progress in a sequential fashion
- concurrent processes
  - execution of a process may progress in a parallel fashion
  - reasons for allowing concurrent execution
    - resource sharing (physical and logical)
    - computation speedup (if there are multiple processing elements)
    - modularity
    - convenience
- independent processes
  - cannot affect or be affected by the other processes executing in the system
    - state is not shared
    - deterministic execution (depends solely on input)
    - reproducible execution
    - can be stopped and restarted (no ill effects)
- cooperating processes
  - can affect or be affected by the other processes executing in the system
    - state is shared
    - result is unpredictable (depends on relative execution sequence)
    - nondeterministic execution
- heavyweight processes (tasks with one thread - UNIX)
- lightweight processes (threads - basic unit of CPU utilization)
  - task is an environment in which threads execute
    - threads share: code, address space, OS resources — less expensive context switches
    - threads own: register space, stack

The extensive sharing makes CPU switching among peer threads and threads' creations inexpensive.
Threads can be supported by:

- Kernel - set of system calls (Mach, OS/2)
- Above the kernel - set of library calls at the user level (Andrew)

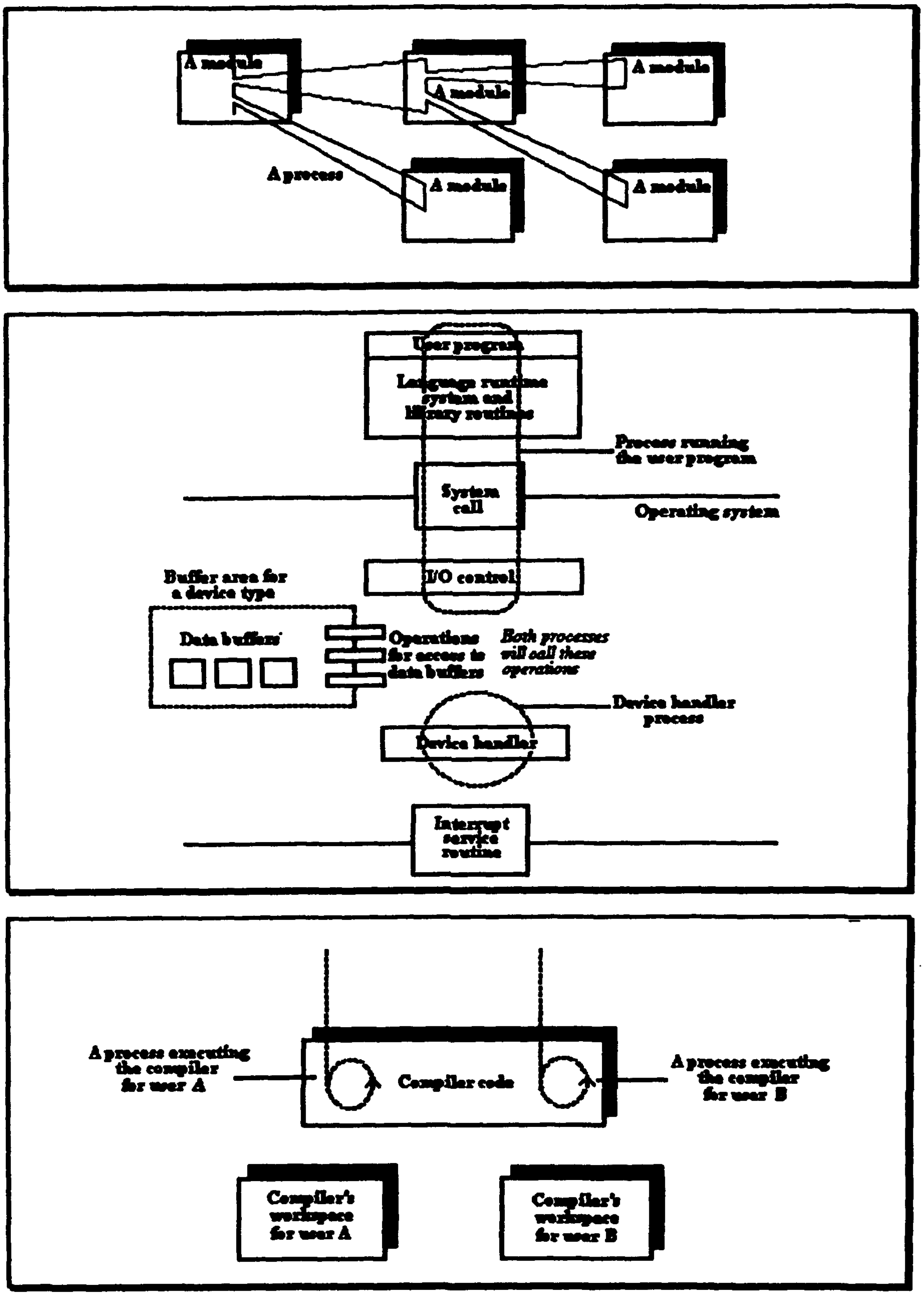


Figure 2.1. Examples of the processes

Process Model

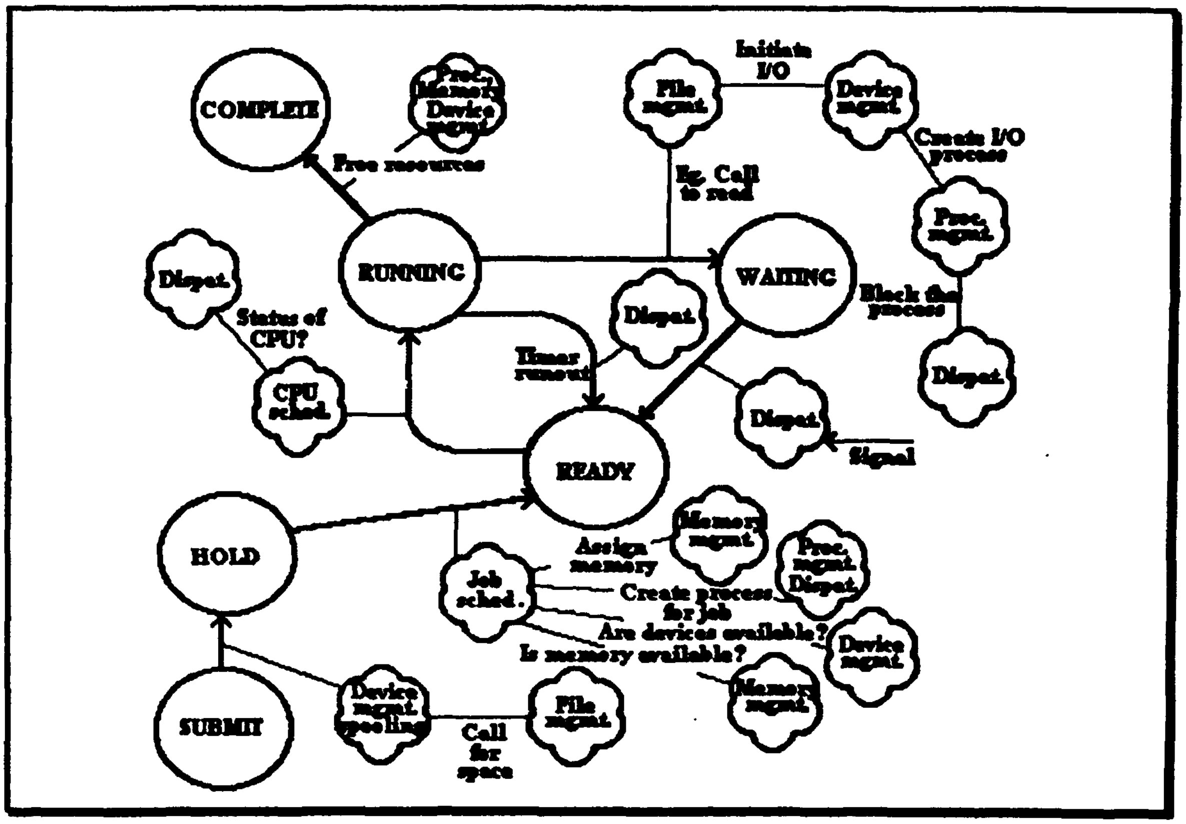


Figure 2.2. Process State Model

- Process states (names are arbitrary and vary from system to system)
  - submit
  - hold
  - ready ● waiting to be assigned to a processor
  - running ● instructions are being executed
  - waiting ● for some event to occur: I/O completion
  - complete

Implementation of Processes

Data structures

- Process (Task) Control Block (record)
  - state
  - pointer
  - program counter (PC)
  - CPU registers (accumulators, index registers, stack pointers, general-purpose registers)
  - process ID
  - scheduling information
    - priority,
    - pointers to scheduling queues,
    - other scheduling parameters
  - memory-management information
    - limit registers,
    - Page Table Base Register,...
  - accounting information
    - CPU time used,
    - real time used,
    - time limits,
    - account number,
    - job or process numbers,...

- ⓘ I/O status information
  - outstanding I/O requests,
  - I/O devices allocated to this process,
  - list of open files,...
- ⓘ identity of children processes

ⓘ EXAMPLE - some of the more common fields presented in UNIX

Process Management	Memory Management	File Management
Registers	Pointer to text segment	UMASK mask
Program counter	Pointer to data segment	Root directory
Program status word	Pointer to bss segment	Working directory
Stack pointer	Exit status	File descriptors
Process state	Signal status	Effective uid
Time when process started	Process id	Effective gid
CPU time used	Parent process	System call parameters
Children's CPU time	Process group	Various flag bits
Time of next alarm	Real uid	
Message queue pointers	Effective uid	
Pending signal bits	Real gid	
Process id	Effective gid	
Various flag bits	Bit maps for signals	
	Various flag bits	

ⓘ EXAMPLE - high-level language (Ada) description of the process descriptor structure

```
type STATE is (SUBMIT, HOLD, READY, RUNNING, WAITING, COMPLETE);
type DURATION is new REAL;
type ADDRESS is access LONG_INTEGER;
type PCB is
    record
        -- process management
        PROCESS_ID: INTEGER;
        PROCESS_STATE: STATE;
        PC: LONG_INTEGER;
        ACC: LONG_INTEGER;
        -- and other registers
        SP: ADDRESS;
        MESS_QUEUE: ADDRESS;
        CPU_TIME: DURATION;
        -- etc
        -- memory management
        TEXT_SEG: ADDRESS;
        DATA_SEG: ADDRESS;
        -- etc
        -- file management
        ROOT_DIR: STRING(1..256);
        WORK_DIR:STRING(1..256);
        FD: ADDRESS;
        -- etc
    end record;
```

ⓘ The procedure of saving the state of a process and setting up the state of another is called **context switching**. The instructions that are executed in performing these operations and the frequency at which context switching happens are an overhead at the lowest level of a system.

Functions

ⓘ The creation and deletion of processes

- ⓘ process creation
  - via **submit** OS creates new processes for a job
  - via ***create_process*** system call (dynamically during the execution of process) (UNIX: *fork*)
    - ⓘ creating processes: **parent**
    - ⓘ created processes: **children** (UNIX: copy of the address space of the parent)
      - children may create their children
      - **resources:** inherited from parent or directly from OS ⓤ overloading
      - **execution:** parent concurrently with children or parent waits until all children terminate
- ⓘ process termination
  - via ***terminate_process*** system call (after the last statement)
    - ⓘ return data to parent process
    - ⓘ ask OS to delete process

via *terminate_process* system call (*abort* from another process: usually parent or OS)

- reasons: exceeded usage of resources
  - task assigned to the child is no longer required
  - parent is terminated ● cascading termination initiated by the OS (usually children cannot exist after parent finishes)
- needed information
  - ID of the children processes
  - state of the children processes

- The suspension and resumption of processes
- The provision of mechanisms for process synchronization
- The provision of mechanisms for process communication
- The provision of mechanisms for deadlock handling

Lecture Three

Textbook

Abraham Silberschatz, James L. Peterson, Peter B. Galvin
"Operating System Concepts", Addison-Wesley, third edition, 1991, Part 2, Chapter 5 (5.1 - 5.4)

References

- Andrew S. Tanenbaum
  "Modern Operating Systems", Prentice-Hall, 1992, Part 1, Chapter 2 (2.2.1 - 2.2.5)
- Jean Bacon
  "Concurrent Systems - An Integrated Approach to Operating Systems, Database, and Distributed Systems", Addison-Wesley, 1993, Chapters 7, 9
- Narain Gehani
  "Ada: Concurrent Programming", Silicon Press, 1991, Chapter 2

Goals

- To show high-level language support to the concept of the process
- To state the problems of the coordination and synchronization of concurrent activities
- To analyze some of the programming solutions to the critical-section problem
- To discuss hardware support for synchronization

Content

- Tasking and Processes (1 hour)
  - Language System Support for Concurrency
- Process Coordination and Synchronization (2 hours)
  - The Critical-Section Problem
  - Synchronization Hardware
  - Semaphores

Lecture Three

Language System Support for Concurrency

- **Concurrent systems built from sequential programs with system calls**
  "Each unit of the concurrent system may be a single sequential process and the operating system may provide system calls to allow interaction with other processes in addition to system calls for requesting operating system service." "A major problem with the approach is portability of the concurrent system. The processes have made use of an operating system interface (a number of system calls) that has been provided to allow interactions between processes. If the system is to be able to run on any other operating system it is necessary for the same interface to be present. Syntactic differences can be allowed for when the code is ported, but semantic differences between operating systems are likely to occur in this area." Jean Bacon, "Concurrent Systems - An Integrated Approach to Operating Systems, Database, and Distributed Systems", Addison-Wesley, 1993, p. 166

- **Coroutines (Modula-2, BCPL)**
  Independent subprograms within a single program:
  - **shared data**
  - **private data (own stack)**
  - **instructions to create and delete a coroutine**
  - **instruction to suspend coroutine execution temporarily but retain its state**
  - **instruction to pass control explicitly from the suspending coroutine to another**
  - **instruction to return control to the caller**
  - **coroutine activations must be scheduled at the user-level**
  - **single thread of control**
  - **no possibility for immediate response**
  - **transfer of control between coroutine activations involves very little overhead**

- **Language-level Processes (Concurrent Pascal, occam)**
  Unlike coroutines, control is managed by an outside agency:
  - Operating system supports one process for one program: The processes within the program are managed internally by the language runtime system which effectively re-implements a process scheduler. The application programmer does not have to program the transfer of control between the language-level processes. Problem: if language-level process makes a system call to the OS to do I/O and becomes blocked then no other subprocess in the program can run.
  - Operating system recognizes subprocesses defined in a program: They become operating system processes (and may be called threads) and are scheduled by the operating system to run on processors concurrently.

- **Ada Tasking Model**
  Tasks become active just prior to the first executable statement following the declarations in an unspecified order. Task is completed after the execution of its body or if the exception is raised for which a handler has not been provided. A task terminates if its execution is completed and all its dependent tasks have terminated, or if it is waiting at a **terminate** alternative and all of its dependent tasks have terminated, there are no outstanding entry calls, its master has completed execution, all dependent tasks of the master have either terminated or are waiting at a **terminate** alternative.
  - Task specification

```
task [type] identifier [ is
        entry declarations
        [representation clauses]          ⇐ associating interrupts with entry calls
end identifier ];
```

  - Task body

```
task body identifier is
        declaration
begin
        statements
[ exception
        exception handlers ]
end identifier;
```

  - Entry declarations
    Rendezvous: matching accept statements
    Associated with each entry of a task is a queue where all newly arriving entry calls are inserted and accepted by the task in FIFO order.

```
entry signal;                         ⇐ synchronization
entry set(T: in duration);
entry read(C: out character);
entry request(ID)(D: in out data):    ⇐ family of entries; ID is discrete type: ID'FIRST .. ID'LAST

accept signal;
accept set(T: in duration) do
        period := T;
end set;
accept request(ID)(D: in out data) do
        -- statements
end request;
```

- Entry calls

  calling task blocked while waiting for the call to be accepted and for the duration of rendezvous

```
disk.request(4)(x);
sem.signal;
```

- Select statement

```
select
      [ when condition1 => ] alternative1
or
      ...
or
      [ when conditionN => ] alternativeN
[ else statements ]
end select;
```

A select alternative can have one of the following forms:

```
accept_statement; [ statements ]
delay t; [ statements ]
terminate;
```

Process Coordination and Synchronization

- Potential concurrent execution of
  - operating system processes
  - user processes
- Mechanisms for orderly execution
  - process synchronization
  - process communication
- Example of the potentially concurrent program - Producer-Consumer Class of Problems
  - A producer process produces information that is consumed by a consumer process. To allow concurrent execution, buffer must be created to be filled by the producer and emptied by the consumer. Buffer may be unbounded or bounded.

Producer	Buffer	Consumer
print program	characters	printer driver
compiler	assembly code	assembler
assembler	object modules	loader

- Erroneous solution of the bounded-buffer producer-consumer problem

```
procedure PRODUCER_CONSUMER is
      N: constant INTEGER range 0..1 := 1000;
      I, O: INTEGER range 0..N-1 := );
      COUNTER: INTEGER := 0;
      -- shared variable for two tasks - mutually exclusive access is necessary!

      BUFFER: array (INTEGER range 0..N-1) of ITEM;
      -- synchronized access is necessary!

      task PRODUCER;
      task body PRODUCER is
            NEXTP: ITEM;
      begin
            loop
                  PRODUCE(NEXTP);
                  while COUNTER = N loop null; end loop;
                  -- synchronization: cannot produce items if buffer is full
                  BUFFER(I) := NEXTP;
                  I := (I + 1) mod N;
                  COUNTER := COUNTER + 1;
                  -- without mutual exclusion program is erroneous
            end loop;
      end PRODUCER;

      task CONSUMER;
      task body CONSUMER is
            NEXTC: ITEM;
```

```
begin
    loop
        while COUNTER = 0 loop null; end loop;
        -- synchronization: cannot consume items if buffer is empty
        NEXTC := BUFFER(O);
        O := (O + 1) mod N;
        COUNTER := COUNTER - 1;
        -- without mutual exclusion program is erroneous
        CONSUME(NEXTC);
    end loop;
end CONSUMER;

begin
    null;
end PRODUCER_CONSUMER;
```

The Critical-Section Problem

A critical section is a segment of code in which the process may be changing common variables. A solution to the problem must satisfy the following requirements:

- mutual exclusion
- progress:"If no process is executing in its critical section and there exist some processes that wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in the decision as to which will enter its critical section next, and this selection cannot be postponed indefinitely." A. Silberschatz, J.L. Peterson, P.B. Galvin:"Operating System Concepts", third edition, Addison-Wesley 1990, p. 134
- bounded waiting

Two-process solutions

- First solution (erroneous)

```
procedure ALGORITHM1 is
    TURN: INTEGER range 0..1 := 0;

    task P0;
    task body P0 is
    begin
        loop
            while TURN <> 0 loop null; end loop;
                -- critical section
            TURN := 1;
                -- remainder section
        end loop;
    end P0;

    task P1;
    task body P1 is
    begin
        loop
            while TURN <> 1 loop null; end loop;
                -- critical section
            TURN := 0;
                -- remainder section
        end loop;
    end P1;

begin
    null;
end ALGORITHM1;
```

The execution sequence which does not satisfy progress requirement:

1. TURN is initially 0
2. Task P_0 enters and exists its critical section and sets TURN to 1
3. Task P_0 is interrupted in its remainder section
4. Task P_1 enters and exists its critical section, sets TURN to 0, enters and exists its remainder section and wishes to enter its critical section again
5. Task P_1 has to wait although task P_0 is in its remainder section, because TURN is 0

Second solution (erroneous)

```
procedure ALGORITHM2 is
    FLAG: array (INTEGER range 0..1) of BOOLEAN := (FALSE, FALSE);

    task P0;
    task body P0 is
    begin
        loop
            FLAG(0) := TRUE;
            while FLAG(1) loop null; end loop;
                -- critical section
            FLAG(0) := FALSE;
                -- remainder section
        end loop;
    end P0;

    task P1;
    task body P1 is
    begin
        loop
            FLAG(1) := TRUE;
            while FLAG(0) loop null; end loop;
                -- critical section
            FLAG(1) := FALSE;
                -- remainder section
        end loop;
    end P1;


begin
    null;
end ALGORITHM2;
```

The execution sequence which may lead to indefinite looping of tasks P_0 and P_1 in their respective **while** statements:

1. Task P_0 is interrupted after assigning TRUE to FLAG(0)
2. Task P_1 is interrupted after assigning TRUE to FLAG(1)
3. Both tasks may proceed now but they will endlessly loop within their **while** statements

Third solution

```
procedure ALGORITHM3 is
    TURN: INTEGER range 0..1 := 0;
    FLAG: array (INTEGER range 0..1) of BOOLEAN := (FALSE, FALSE);

    task P0;
    task body P0 is
    begin
        loop
            FLAG(0) := TRUE;
            TURN := 1;
            while (FLAG(1) and TURN = 1) loop null; end loop;
                -- critical section
            FLAG(0) := FALSE;
                -- remainder section
        end loop;
    end P0;

    task P1;
    task body P1 is
    begin
        loop
            FLAG(1) := TRUE;
            TURN := 0;
            while (FLAG(0) and TURN = 0) loop null; end loop;
                -- critical section
            FLAG(1) := FALSE;
                -- remainder section
        end loop;
    end P1;
```

```
begin
    null;
end ALGORITHM3;
```

Multiple-process solution

- Bakery algorithm

Synchronization Hardware

- Disallow interrupts
- Atomic execution
  - *test_and_set*
  - *swap*

Semaphores

wait(s) (or P(s)):: **while** s ⊙ 0 **loop null end loop**; s := s - 1; (modify without busy waiting)
signal(s) (or V(s)):: s := s + 1;

- synchronization

```
        synch := 0;
P1::    S1;
        signal(synch);
P2::    wait(synch);
        S2;
```

- mutual exclusion

```
loop
    wait(mutex);
        -- critical section
    signal(mutex);
        -- remainder section
end loop;
```

- advantages
  avoid busy waiting; process blocks itself ⇒ waiting queue associated with the semaphore (policy: LIFO ● starvation)

- disadvantages
  - rely to much on programmer
  - deadlock
  - unreadable
  - complex

Lecture Four

Textbook

Abraham Silberschatz, James L. Peterson, Peter B. Galvin
"Operating System Concepts", Addison-Wesley, third edition, 1991, Part 2, Chapter 5 (5.5 - 5.7)

References

- Andrew S. Tanenbaum
  "Modern Operating Systems", Prentice-Hall, 1992, Part 1, Chapter 2 (2.2.6 - 2.2.9, 2.3)
- Jean Bacon
  "Concurrent Systems - An Integrated Approach to Operating Systems, Database, and Distributed Systems", Addison-Wesley, 1993, Chapters 10 -12, 14

Goals

- To analyze solutions to some of the classical problems of synchronization
- To present different language constructs
- To analyze some of the programming solutions to the critical-section problem
- To discuss principles of IPC
- To discuss main concepts of scheduling

Content

- Process Coordination and Synchronization (2 hours)
  - Classical Problems of Synchronization
  - Language Constructs and Interprocess Communication
- Scheduling (1 hour)
  - Concepts

Classical Problems of Synchronization

The Bounded-Buffer Problem

① Ada 83 solution

```
procedure PRODUCER_CONSUMER is

    task PRODUCER;
    task body PRODUCER is
        NEXTP: ITEM;
    begin
        loop
            PRODUCE(NEXTP);
            BUFFERING.PUT(NEXTP);
        end loop;
    end PRODUCER;

    task CONSUMER;
    task body CONSUMER is
        NEXTC: ITEM;
    begin
        loop
            BUFFERING.GET(NEXTC);
            CONSUME(NEXTC);
        end loop;
    end CONSUMER;

    task BUFFERING is
        entry PUT(X: in ITEM);
        entry GET(X: out ITEM);
    end;
    task body BUFFERING is
        N: constant INTEGER range 0..1 := 1000;
        I, O: INTEGER range 0..N-1 := );
        COUNTER: INTEGER := 0;
        BUFFER: array (INTEGER range 0..N-1) of ITEM;
    begin
        loop
            select
                when COUNTER < N =>
                    accept PUT(X: in ITEM) do
                        BUFFER(I) := X;
                    end;
                    I := (I + 1) mod N;
                    COUNTER := COUNTER + 1;
            or
                when COUNTER > 0 =>
                    accept GET(X: out ITEM) do
                        X := BUFFER(O);
                    end;
                    O := (O + 1) mod N;
                    COUNTER := COUNTER - 1;
            end select;
        end loop;
    end BUFFERING;

begin
    null;
end PRODUCER_CONSUMER;
```

② Ada 9X solution (S. Tucker Taft: "Ada 9X: A Technical Summary", CACM, November 1992, Vol.35, No. 11)

```
generic
    type MESSAGE_TYPE is private;
package MAILBOX_PKG is
    type MESSAGE_ARRAY is array(POSITIVE range <>) of MESSAGE_TYPE;
    protected type MAILBOX(SIZE: NATURAL) is
        function COUNT return NATURAL;
        procedure DISCARD_ALL;
        entry PUT(MESSAGE: in MESSAGE_TYPE);
        entry GET(MESSAGE: out MESSAGE_TYPE);
```

```
    private
        CONTENTS: MESSAGE_ARRAY(1..SIZE);
        CURRENT_COUNT: NATURAL := 0;
        PUT_INDEX: POSITIVE := 1;
        GET_INDEX: POSITIVE := 1;
    end MAILBOX_PKG:
package body MAILBOX_PKG is
    protected body MAILBOX is
        function COUNT return NATURAL is
        begin
            return CURRENT_COUNT:
        end COUNT:

        procedure DISCARD_ALL is
        begin
            COUNT := );
            PUT_INDEX := 1;
            GET_INDEX := 1;
        end DISCARD_ALL;

        entry PUT(MESSAGE: in MESSAGE_TYPE) when COUNT < SIZE is
        begin
            CONTENTS(PUT_INDEX) := MESSAGE;
            PUT_INDEX := PUT_INDEX mod SIZE + 1:
            COUNT := COUNT + 1;
        end PUT;

        entry GET(MESSAGE: out MESSAGE_TYPE) when COUNT > 0 is
        begin
            MESSAGE := CONTENTS(GET_INDEX);
            GET_INDEX := GET_INDEX mod SIZE + 1:
            COUNT := COUNT - 1;
        end GET;
    end MAILBOX;
end MAILBOX_PKG;

with TEXT_IO, MAILBOX_PKG;
procedure TEST is
    type LINE is
        record
            LENGTH: NATURAL := 0;
            DATA: STRING(1..80);
        end record;
    package LINE_BUFFER_PKG is
        new MAILBOX_PKG(MESSAGE_TYPE => LINE);
    LINE_BUFFER: LINE_BUFFER_PKG.MAILBOX(SIZE => 20);

    task PRODUCER;
    task body PRODUCER is
        L: LINE;
    begin
        for I in 1..100 loop
            TEXT_IO.GET_LINE(L.DATA, L.LENGTH);
            LINE_BUFFER.PUT(L);
        end loop;
    end PRODUCER;

    task CONSUMER;
    task body CONSUMER is
        L: LINE; C: NATURAL;
    begin
        for I in 1..100 loop
            LINE_BUFFER.GET(L);
            TEXT_IO.PUT_LINE(L.DATA(1..L.LENGTH));
            C := LINE_BUFFER.COUNT(L);
            if C > LINE_BUFFER / 2 then
                TEXT_IO.PUT_LINE("Buffer count now=" & INTEGER'IMAGE(C));
            end if;
        end loop;
    end CONSUMER;
```

```
begin
    null;
end TEST;
```

The Readers and Writers Problem

The Dining-Philosophers Problem

```
with TEXT_IO; use TEXT_IO;
procedure DINING is
    package IO_INT is new INTEGER_IO(INTEGER); use IO_INT;
    subtype ID is INTEGER range 1..5;

    task type PHILOSOPHER is
        entry GET_ID(J: in ID);
    end PHILOSOPHER;

    task type FORK is
        entry PICK_UP;
        entry PUT_DOWN;
    end FORK;

    F: array(ID) of FORK;
    P: array(ID) of PHILOSOPHER;

    task body PHILOSOPHER is
        ME, LEFT, RIGHT: ID;
        LIFE_LIMIT: constant 100_000;
        TIMES_EATEN: INTEGER := 0;
    begin
        accept GET_ID(J: in ID) do M := J; end GET_ID;
        LEFT := ME; RIGHT := ME mod 5 + 1;
        while TIMES_EATEN /= LIFE_LIMIT loop
            THINK;
            F(LEFT).PICK_UP; F(RIGHT).PICK_UP;
            EAT;
            F(RIGHT).PUT_DOWN; F(LEFT).PUT_DOWN;
            TIMES_EATEN := TIMES_EATEN + 1;
        end loop;
    end PHILOSOPHER;

    task body FORK is
    begin
        loop
            select
                accept PICK_UP; accept PUT_DOWN;
            or
                terminate;
            end select;
        end loop;
    end FORK;

begin
    for K in ID loop P(K).GET_ID(K); end loop;
end DINING;
```

The Sleeping-Barber Problem

Language Constructs and Interprocess Communication

- Conditional Critical Regions
  Compiler should check and enforce that
  - shared data is only accessed from within a critical region
  - critical regions are entered and left correctly by processes

  Syntax:
  - shared as an attribute of any data type
  - a region declaration

    region <shared data> **when** <condition> **begin** <statements> **end**

- **Monitors**

A monitor has the structure of an abstract data object

- Encapsulated data are shared
- Each operation is executed under mutual exclusion
- Process may delay itself on condition variable

Syntax:

monitor name **is**
    entry procedures <list of procedure names visible externally>
    variable declarations and initialization of values
    external procedure declarations and bodies
    internal procedure declarations and bodies
end name;

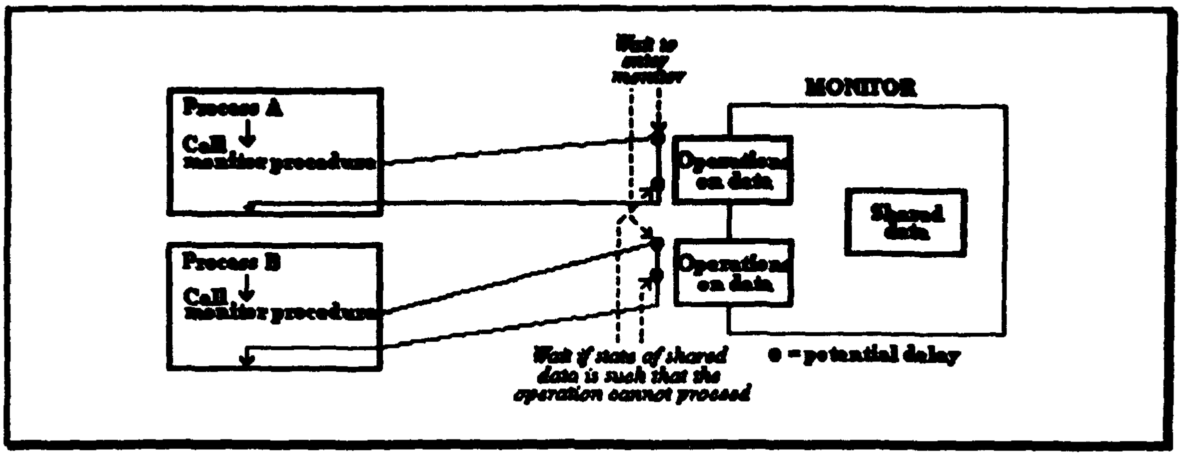


Figure 4.1. Program structure with monitors

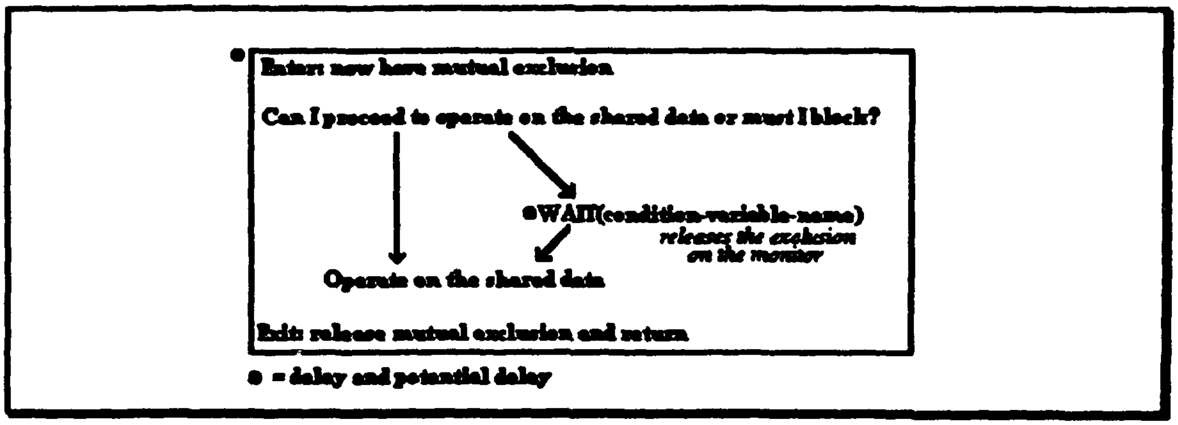


Figure 4.2. Condition synchronization under exclusion in a monitor procedure

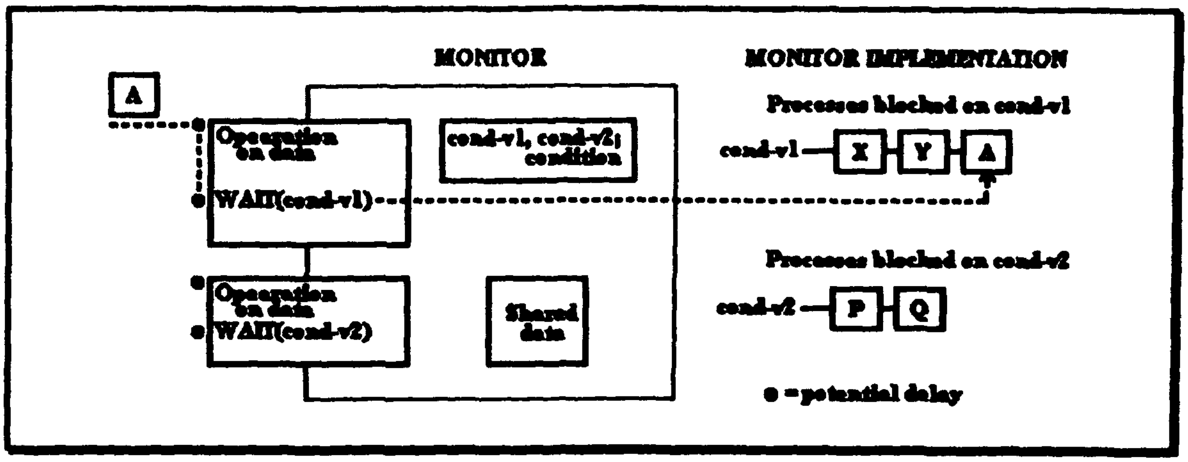


Figure 4.3. Condition variable queues

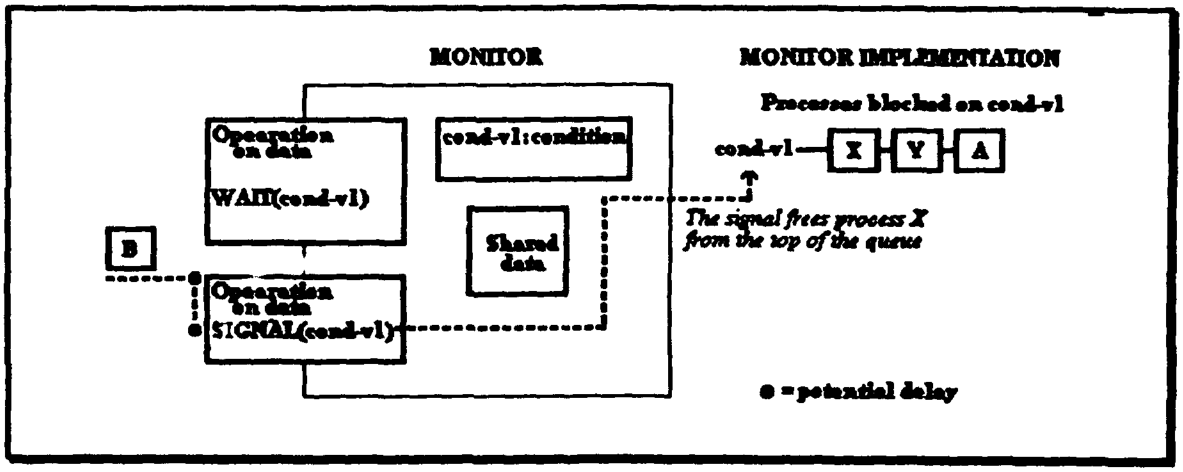


Figure 4.4. Signalling a condition variable

Who will be blocked after the SIGNAL operation?

- signalling process: signalling process will be put at the head of the monitor wait queue → monitor data must be in a consistent state before a SIGNAL is executed
- signalled process: transfer the signalled process from the condition variable queue on which it is waiting to the head of the queue of processes waiting to enter the monitor

- Path Expressions

- Message Passing Mechanisms
  - Asynchronous
    - Receiving from "anyone"
    - Request and reply primitives
    - Multiple ports per process
    - Input ports, output ports and channels
    - Global ports
    - Broadcast and multicast
    - Message forwarding
  - Synchronous
    - occam channel
    - Linda abstraction

Scheduling Concepts

multiprogramming ● maximize CPU utilization
throughput ● amount of work accomplished in a given time interval
enter the system → job queue (on mass storage awaiting allocation of main memory) → enter the main memory → ready queue → running → request of I/O → device queue

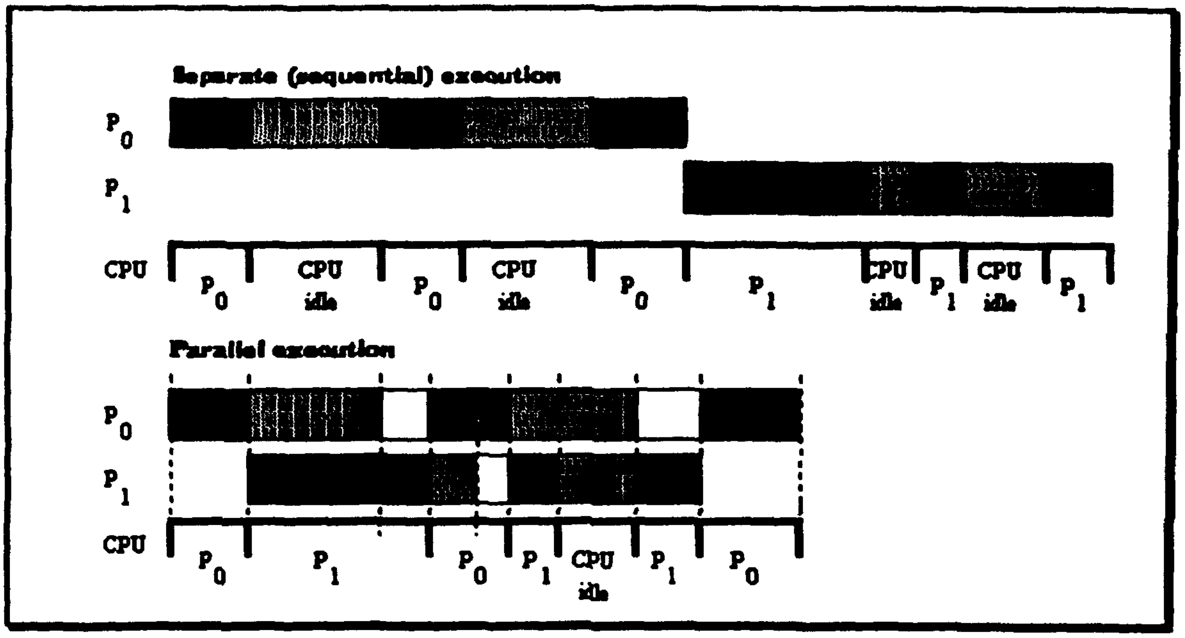


Figure 4.5. Benefits of the parallel execution

Select processes from various scheduling queues

⑦ long-term scheduler (job scheduler) ● may not exist

Job queue is on the mass storage device
Choose a small subset of jobs submitted and lets them into the system
Creates processes
Assigns some resources
Requirements: executes less frequently
controls the degree of multiprogramming (*if stable same rate of creation and deletion*)
good process mix (I/O-bound and CPU-bound)

⑦ medium-term scheduler

Swapping
Partially *executed* queue ● secondary memory
Requirements: improve the process mix
reason - change of memory requirements

⑦ short-term scheduler (CPU scheduler)

CPU scheduler manages ready queue
When CPU scheduling takes places?
⑦ running state ● waiting state
Ex. I/O request, wait for termination of the child process
process is blocked U another should be selected
⑦ running state ● ready state
Ex. interrupt (time-out,...)
after OS services (interrupt routine,...) finish their job U another process (possibly the same) should be selected for execution
⑦ waiting state ● ready state
Ex. completion of I/O
⑦ running state ● complete state
Process consists of a cycle of CPU execution and I/O wait
⑦ CPU-bound process
⑦ I/O-bound process
Functionality
1. Choose a process for execution from ready queue
2. Call dispatcher to do the physical assignment of the job to the processor
ready queue (main memory)
assigns processor to a process: which, when and for how long?
Requirements: executes very often ● must be very fast
statistics on CPU-burst - I/O-burst cycle may be important in selecting an algorithm ● expectations: a very large number of very short CPU-bursts
trigger - change of process state
Schemes: non-preemptive scheduling
trigger: running to waiting state
termination

overhead is smaller, simple, easy to implement, may be reasonable for dedicated systems (ex. database systems) where master process knows how long child will run

preemptive scheduling

trigger: running to ready state (interrupt, time slice)

waiting to ready state (completion of I/O)

circumstances may be changed and more appropriate decision made

context switch

- saving the state of the old process
- loading the saved state for the new process
- context-switch time (● pure overhead) depends on
  - hardware support
    - memory speed
    - number of registers
    - register sets
  - software support
    - special instruction to load/store all registers

dispatcher

- physically gives control to the selected process
- functions
  - switching context
  - switching to user mode
  - jumping to the proper location to restart the program

comparison criteria

Criteria selection defined according to the relative importance of these measures

- CPU utilization (% of time when CPU is busy) — maximize
- throughput (number of proceses per time unit) — maximize

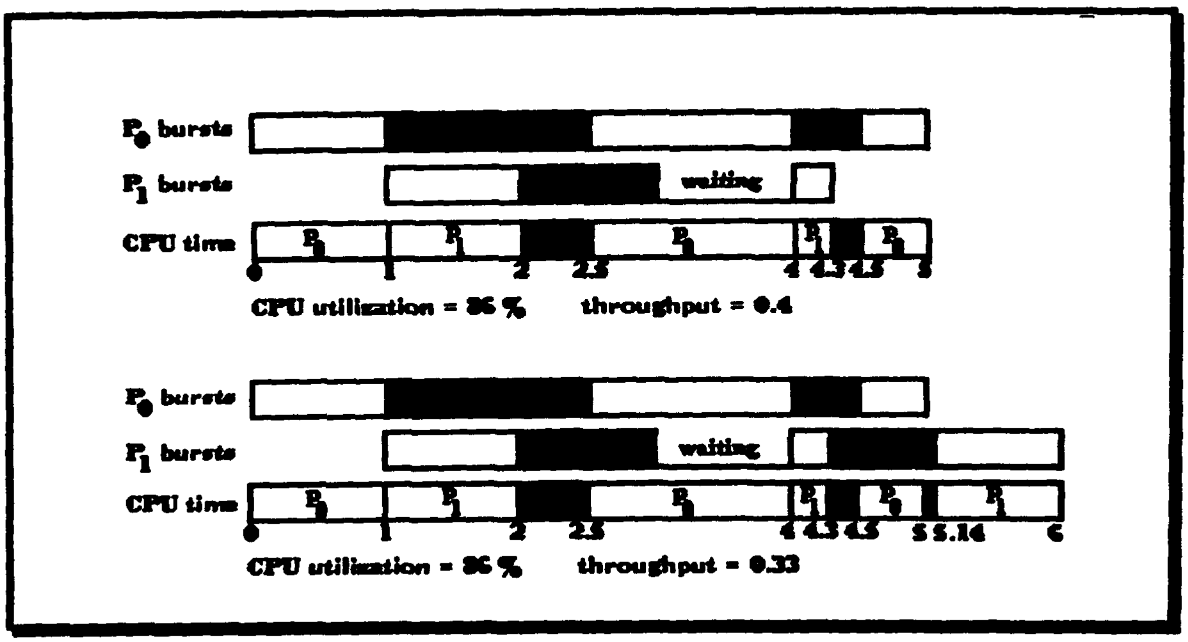


Figure 4.6. Difference between two comparison criteria

- turnaround time (time of completion - time of submission) — minimize
- waiting time (in ready queue) — minimize
- response time (time of first response - time of submission) — minimize

or

optimize the min and max values rather then average

or

optimize average measure

or

minimize variance

- fairness (make sure each process gets its fair share of the CPU)

"Any scheduling algorithm that favors some class of jobs hurts another class of jobs.",

Lecture Four

Kleinrock L.: "Queuing Systems", Vol. I, John Wiley 1974

Evaluation

- analytic evaluation
  - algorithm
  - system workload ⇒ performance for that workload
  - Deterministic modelling - predetermined workload
  - Queuing modelling (queuing-network analysis)
    - measure distribution of I/O and CPU bursts
    - arrival rates
    - service rates
- simulation (programming a model of the computer system)
  - clock = variable
  - clock value increased ⇒ system state modified to reflect the activities of the devices, the processes and the scheduler
  - statistics of algorithm performance
  - random number generator (for events according to the probability distributions) or trace tapes
- implementation

Analyze the real situation of CPU utilization taking into account time for OS intervention

Lecture Five

Textbook

Abraham Silberschatz, James L. Peterson, Peter B. Galvin
"Operating System Concepts", Addison-Wesley, third edition, 1991, Part 2, Chapter 4 (4.4 - 4.7), Chapter 6 (6.1 - 6.2)

References

- Andrew S. Tanenbaum
  "Modern Operating Systems", Prentice-Hall, 1992, Part 1, Chapter 2 (2.4), Chapter 6 (6.1 - 6.2)
- Jean Bacon
  "Concurrent Systems - An Integrated Approach to Operating Systems, Database, and Distributed Systems", Addison-Wesley, 1993, Chapter 6 (6.6 - 6.8), Chapter 16 (16.1 - 16.5)

Goals

- To analyze some of the existing scheduling algorithms
- To present different evaluation techniques
- To introduce the concept of deadlock

Content

- Scheduling and Dispatch (2 hours)
  - Scheduling Algorithms and Their Evaluation
- Deadlocks (1 hour)
  - System Modelling
  - Deadlock Characterisation

Scheduling Algorithms and Their Evaluation

Algorithms

- **First-Come First-Served (FCFS)**
  - FIFO queue
    - incoming ready process is linked onto the tail of the ready queue
    - CPU is allocated to the process at the head of the ready queue
  - Average waiting time is long
  - Convey effect — Ex. one CPU-bound process and many I/O-bound processes
  - Nonpreemptive

- **Shortest-Job-First (SJF)**
  - Criteria
    - process with smallest next CPU-burst
    - tie - FCFS
  - Provably optimal (minimum average waiting time)

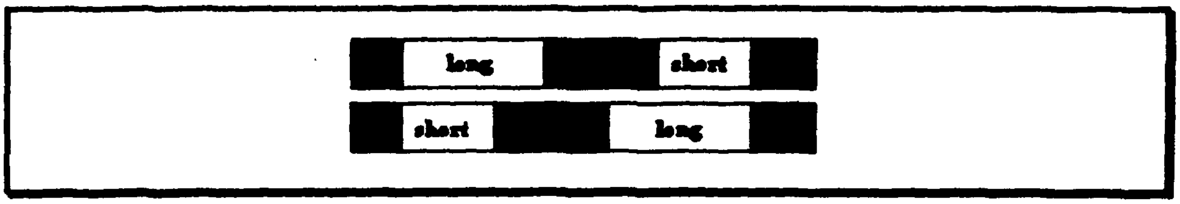


Figure 5.1. SJF - Optimal Algorithm

- Problems
  - How can CPU know the length of the next CPU-burst interval?
  - batch systems
    - job scheduler
    - user should submit punched control card with process time limit ("time-limit-exceeded error")
  - prediction
    - Exponential average: $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$, $0 \le \alpha \le 1$
      - t_n - recent history, τ_n - past history
      - $\alpha = 0$, past history prevails
      - $\alpha = 1/2$, equally distributed
      - $\alpha = 1$, recent history prevails
    - Expanded formula
      - $\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \dots + (1 - \alpha)^j \alpha t_{n-j} + \dots + (1 - \alpha)^n \tau_0$
      - τ_0 - constant or overall system average
- Type
  - preemptive
    - $\text{CPU-burst}_{\text{next process}} < \text{CPU-burst}_{\text{what is left of the currently executing process}}$
    - shortest-remaining-time-first
  - nonpreemptive
- Example

```
task SJF_SCHEDULER is
      entry ADD(JOB:ID; T:DURATION);
      entry GET(JOB: out ID);
      -- return the next job to be executed and delete it
end SJF_SCHEDULER;

task body SJF_SCHEDULER is
      I:ID;
      PERIOD:DURATION;
begin
      loop
            select
                  accept ADD(JOB:ID; T:DURATION) do
                        I := JOB;
                        PERIOD := T;
                  end ADD;
                  INSERT(I, PERIOD);
            or
```

```
            when not EMPTY ⇒
                accept GET(JOB: out ID) do
                    SMALLEST(JOB);
                end GET;
        end select;
    end loop;
end SJF_SCHEDULER;

procedure INSERT(JOB:ID; T:DURATION);
    -- add job to the set; T is the next CPU-burst time
procedure SMALLEST(JOB: out ID);
    -- JOB is set to the id of the job with the smallest next CPU-burst time and this job is deleted from the set; call
    -- SMALLEST only when the set is not empty
function EMPTY return BOOLEAN;
```

- Priority
  - Criteria
  - process with highest priority
  - tie - FCFS
  - SJF - special case p ≈ ♦
  - Assumption: low number ⋆ high priority
  - Priorities
  - defined internally
    - time limits
    - memory requirements
    - number of open files
    - average I/O-burst/average CPU-burst
  - defined externally
    - importance of the process
    - type and amount of funds being paid for computer use
    - political and other factors
  - Problems

  Indefinite blocking (starvation) - steady-stream of high-priority processes

  Solution: aging - gradually increasing the priority of processes
  - Type
  - preemptive: $\text{priority}_{\text{next process}} > \text{priority}_{\text{running process}}$
  - nonpreemptive
- Round-Robin (RR)
  - Criteria
  - time quantum (10-100 ms) (timer set to interrupt after one time quantum)
  - ready queue - circular FIFO
  - average waiting time is long
  - performance depends on the size of the time quantum; very small quantum ⇒ processor sharing
  - Problems: Context switching - solution: time quantum ⇧⇧ context-switch time
  - Type: preemptive
- Multilevel Queue
  - Criteria
  - different ready queues (foreground queue - interactive processes, background queue - batch processes, system processes, interactive
    editing processes,...)
  - process permanently assigned to one queue
  - each queue has its own scheduling algorithm

  - scheduling between the queues: fixed-priority preemptive scheduling or timeslicing
  - Problems: not flexible - process remains permanently into the assigned queue
- Multilevel Feedback Queue
  - Criteria
    - process may move between queues
    - entering queue
    - criteria for upgrade (ex. I/O-bound)
    - criteria when to demote (ex. long waiting time)
- Guaranteed Scheduling

  Make realistic promises about performance and live up to them

  Ex. if there are n users logged in, each will receive 1/n of the CPU power

  System must
    - keep track of how much CPU time a user has had for all his processes since login and how long has been logged in

- compute the amount of CPU each user is entitled to

  $$\frac{\text{time since login}}{n} \quad \text{(dynamic)}$$

- compute

  $$\frac{\text{actual CPU time used}}{\text{CPU time entitled}}$$

- run the process with the lowest ratio
- reschedule when the ratio has moved above its closest competitor

Ex. real-time system: process in greatest danger of missing its deadline

- Multiple Processors
  - issues
    - types of processors
      - heterogeneous system
        own queue
        own scheduling
      - homogeneous systems
        - load sharing
          one queue
        - self-scheduling
          synchronization and mutual exclusion not to pick the same process (common data structure)
        - master-slave structure
          one processor scheduler
          (asymmetric multiprocessing)
          heavy scheduling load ⇒ master processor
          overloaded (bottleneck)
- Conclusion
  Separate scheduling mechanisms and scheduling policies
  Scheduling algorithm is parametrized in some way, but the parameters can be filled in by the user processes (dynamically, rare ?)

Deadlock Concept and Characterization

- System Modelling
  - finite number of resources partitioned into several types consisting of some number of identical instances
    $R = \{(R_1, n_1), (R_2, n_2), ..., (R_m, n_m)\}$
  - finite number of processes
    $P = \{P_1, P_2, ..., P_k\}$
    - request resource (system calls or wait and signal operations)
    - use resource
    - release resource (system calls or wait and signal operations)
  - resource-allocation graph

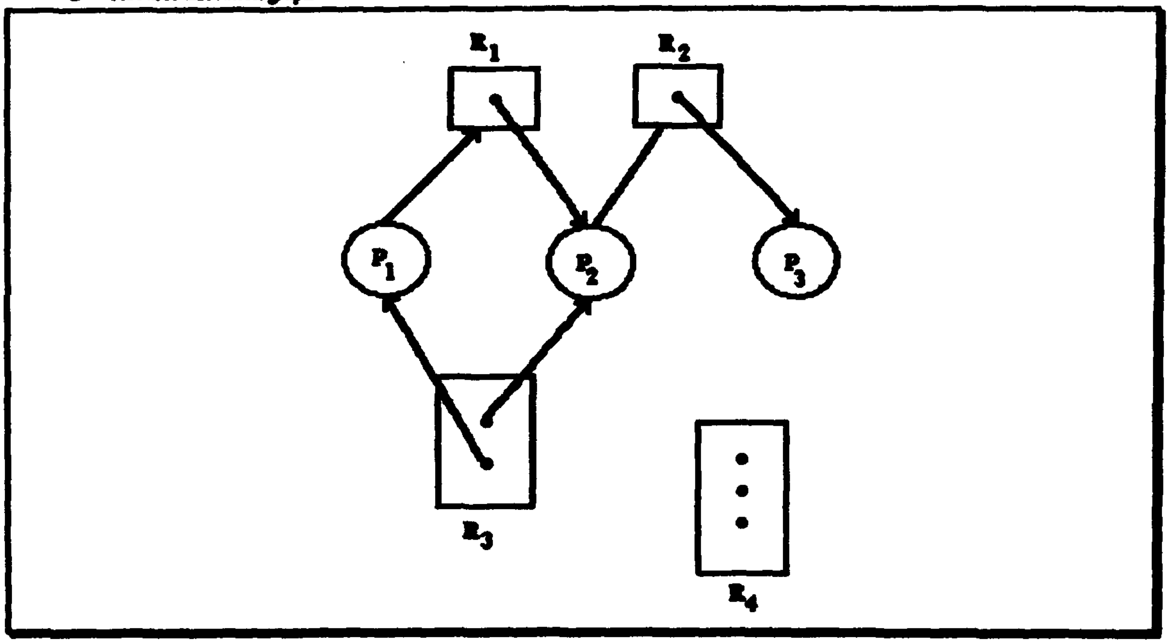


Figure 5.2. Example of the resource-allocation graph

- **Deadlock Characterization**
  A set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set.

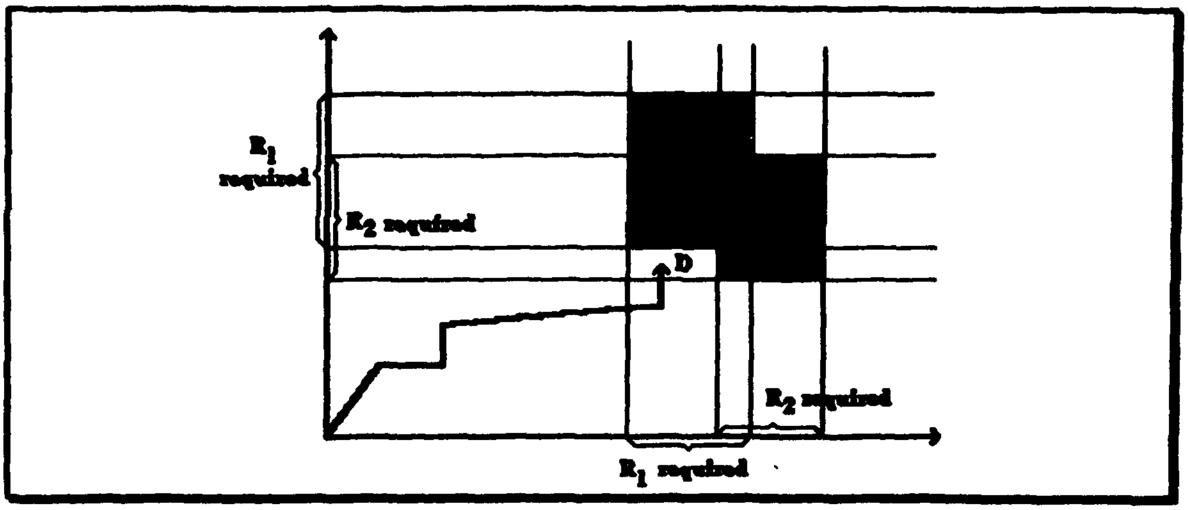


Figure 5.3. Illustration of deadlock

Necessary conditions for deadlock:
- mutual exclusion (at least one resource must be unsharable)
- hold and wait
- no preemption
- circular wait

Situations other than deadlock with no progress:
- livelock (busy waiting on a condition that can never become true)
- starvation (process is indefinitely postponed)

Lecture Six

Textbook

Abraham Silberschatz, James L. Peterson, Peter B. Galvin
"Operating System Concepts", Addison-Wesley, third edition, 1991, Part 2, Chapter 6 (6.3 - 6.7) and Part 3, Chapter 7

References

- Andrew S. Tanenbaum
  "Modern Operating Systems", Prentice-Hall, 1992, Part 1, Chapter 6 (6.3 - 6.6) and Chapter 3 (3.1 - 3.2)
- Jean Bacon
  "Concurrent Systems - An Integrated Approach to Operating Systems, Database, and Distributed Systems", Addison-Wesley, 1993, Chapter 6 (6.2 - 6.6), Chapter 3 (3.1 - 3.2)

Goals

- To introduce different techniques for deadlock handling
- To present concepts of memory management
- To discuss approaches to physical memory management

Content

- Deadlocks (1 hour)
  - Deadlock Handling
- Physical and Virtual Memory Allocation (2 hours)
  - Memory Management

Deadlock Handling

- Ostrich algorithm
- Ensure that the system will never enter a deadlock state
  - deadlock prevention
    Ensure that at least one of the necessary conditions does not hold
    - mutual exclusion must hold for nonsharable resources
    - hold and wait: guarantee that whenever a process requests a resource it does not hold any
      - allocation of all processes prior to execution
      - process must release all resources before requesting another

      Drawbacks: low resource utilization and starvation
    - no preemption: whenever requests cannot immediately be granted process implicitely releases (preempts) all its resources
    - circular wait: total ordering of all resource type with the requirement that each process requests resources in an increasing order of enumeration
  - deadlock avoidance
    - declare the maximum number of resources
    - resource-allocation state is defined by
      - the number of available resources
      - the number of allocated resources
      - maximum demands
    - state is safe if the system can allocate resources to each process up to its maximum in some order and still avoid deadlock
  - Banker's algorithm
- Allow the system to enter a deadlock state and then recover
  - detection algorithm - when it should be invoked?
    - How often is deadlock likely to occur?
    - How many processes will be affected by deadlock when it happens
  - recovery algorithm
    - process termination
      - abort all deadlocked processes
      - abort one process at a time until the deadlock cycle is eliminated
    - resource preemption
      - selecting a victim
      - rollback
      - starvation

Memory Management

- Memory Management without swapping or paging
  - Monoprogramming
  - Multiprogramming and memory usage
  - Multiprogramming with fixed partitions
- Swapping
  - Multiprogramming with variable partitions
  - Bit Maps
  - Linked Lists
  - Buddy system
  - Allocation of Swap Space

Lecture Seven

Textbook

Abraham Silberschatz, James L. Peterson, Peter B. Galvin
"Operating System Concepts", Addison-Wesley, third edition, 1991, Part 3, Chapter 8

References

- Andrew S. Tanenbaum
  "Modern Operating Systems", Prentice-Hall, 1992, Part 1, Chapter 3 (3.3 - 3.8) and Chapter 5 (5.1 - 5.2)
- Jean Bacon
  "Concurrent Systems - An Integrated Approach to Operating Systems, Database, and Distributed Systems", Addison-Wesley, 1993, Chapter 3 (3.1 - 3.3)

Goals

- To introduce concepts of virtual memory
- To discuss a device management through the introductory example of device driver

Content

- Physical and Virtual Memory Allocation (2 hours)
  - Virtual Memory
- Device Management (1 hour)

Virtual Memory

- Demand Paging
  - Page Tables and Inverted Page Tables
  - Paging Hardware
  - Associative Memory
  - Page Replacement
- Page-Replacement Algorithms
  - Optimal Algorithm
  - Not-Recently-Used Algorithm
  - FIFO Algorithm
  - Second Chance Algorithm
  - Clock Algorithm
  - Least Recently Used Algorithm
  - Simulating LRU in software
- Belady's anomaly
- Stack Algorithms
- Predicting page fault rate
- Working Set Model
- Page size
- Allocation of Frames
- Trashing
- Demand Segmentation
- Segmentation with paging

Device Management

- Principles of I/O hardware
  - I/O devices
  - Device Controllers
  - Direct Memory Access (DMA)
- Principles of I/O software
  - goals
  - interrupt handlers
  - device drivers

```
task PRINTER_DRIVER is
    entry PRINT(L: LINE);
    entry READY; --printer ready for next character
    for READY use at 16#80#;
end PRINTER_DRIVER:

task body PRINTER_DRIVER is
    type STATUS_REGISTER is
        record
            INTERRUPT_ENABLED: BOOLEAN;
            CHAIN_RUNNING: BOOLEAN;
        end record;
    for STATUS_REGISTER use
        record
            INTERRUPT INABLED at 0 RANGE 5..5;
            -- map INTERRUPT_ENABLED to bit 5 of first word(0) of storage allocated to
            -- objects of type STATUS_REGISTER
            CHAIN_RUNNING: BOOLEAN;
        end record;
    for STATUS_REGISTER'SIZE use 2;
    -- allocate one word (two bytes per word assumed)
    LINE_LENGTH: constant := 132;
    subtype LINE is STRING(1..LINE_LENGTH);
    PRINTER_REGISTER: STATUS_REGISTER:
    for PRINTER_REGISTER use at 16#3FF40#;
    BUFFER: LINE;
    PRINTER_BUFFER: CHARACTER;
    for PRINTER_BUFFER use at 16#3FF42#;
begin
    PRINTER_REGISTER.INTERRUPT_ENABLED := TRUE;
    PRINTER_REGISTER.CHAIN_RUNNING := FALSE:
```

```
        loop
            select
                accept PRINT(L: LINE) do
                    BUFFER := L;
                end PRINT;
                if not PRINTER_REGISTER.CHAIN_RUNNING then
                    PRINTER_REGISTER.CHAIN_RUNNING := TRUE;
                    delay 1.0;
                end if;
                for I in 1..LINE_LENGTH loop
                    PRINTER_BUFFER := BUFFER(I);
                    accept READY;
                    exit when BUFFER(I) = ASCII.LF or BUFFER(I) = ASCII.FF;
                end loop;
            or
                when PRINTER_REGISTER.CHAIN_RUNNING =>
                    delay 10.0;
                    PRINTER_REGISTER.CHAIN_RUNNING := FALSE;
            end select;
        end loop;
    end PRINTER_DRIVER;
```

- device-independent I/O software
- user-space I/O software

APPENDIX B

CS495 SAFETY CRITICAL SOFTWARE ENGINEERING WITH ADA

ADA CLASS REPORT

Ada Class Report
12 August 1993

STATEMENT OF THE PROBLEM

Most Department of Defense (DoD) contractors currently use the Ada programming language, primarily because DoD mandates its use. As shown by a recent article in *Ada Letters*[1] it is used extensively outside of the defense community:

> Although Ada was originally designed to provide a single flexible yet portable language for real-time embedded systems to meet the needs of the US DoD, its domain of application has expanded to include many other areas, such as large-scale information systems, distributed systems, scientific computation, and systems programming. Furthermore, its user base has expanded to include all major defense agencies of the Western world, the whole of the aerospace community and increasingly many areas in civil and private sectors such as telecommunications, process control and monitoring systems. Indeed, the expansion in the civil sector is such that civil applications now generate the dominant revenues of many vendors.

But too few commercial developers use Ada to make it one of the most popular languages. Part of the reason for this is that new graduates do not come with a knowledge of Ada. Commercial employers who might be considered prime candidates to use Ada are thus faced with additional training time and costs. Worse yet, few in commercial organizations are familiar with the benefits of using Ada. Even safety-critical applications that need Ada the most are often developed by practitioners untrained in software engineering principles[2]:

> The mistakes that were made are not unique to this manufacturer but are, unfortunately, fairly common in other safety-critical systems. As Frank Houston of the US Food and Drug Administration (FDA) said, "A significant amount of software for life-critical systems comes from small firms, especially in the medical device industry; firms that fit the profile of those resistant to or uninformed of the principles of either system safety or software engineering."[3]
>
> Furthermore, these problems are not limited to the medical industry. It is still a common belief that any good engineer can build software, regardless of whether he or she is trained in state-of-the-art software engineering procedures.

IIT teaches many foreign-born students in the Computer Science (CS) Department. Because it is more difficult for foreign-born students to obtain a security clearance, very few obtain jobs with firms doing primarily DoD software development. Therefore, the Ada mandate means little or nothing to these students.

<u>APPROACH TO A SOLUTION</u>

To attract foreign-born students to an Ada class, IIT emphasizes the benefits of using Ada for typical application areas. One such area is safety-critical software. IIT offered a CS 495 course in the summer 1993 term called, "Safety-Critical Software Engineering With Ada." The announcement in Attachment A described some of the benefits of using modern software engineering approaches__including the use of Ada__to this application area.

The course instructor, Fred Francl, surveyed the class at the start of the first class. Attachment B presents the results of this survey of what each student hoped to get out of this class. This survey shows that all but one of the students surveyed mentioned Ada as the major attraction of this course. This was somewhat surprising because the course announcement (Attachment A) listed Object-Oriented (OO) techniques as one of the course features. Despite the current popularity of OO techniques, only two students in the survey listed them as a course priority.

The two-credit course covered the first edition of *Software Engineering With Ada* by Grady Booch. Attachment C shows how each chapter was weighted. The course introduced the Ada features most widely used in DoD projects, offered industry experience with these features, starting in the early eighties and including current experiences. These examples showed how these Ada features helped to integrate modern software engineering principles into the product.

Attachment D presents some of the viewgraphs developed for this course. They were designed to emphasize the advantages of the Ada features being taught. Where appropriate, Ada features were directly compared to features of other popular languages.

The strategy in presenting virtually all Ada features was to give the students the sense that Ada is a general-purpose language. The instructor reinforced this by using Ada in class problems and homework problems to implement solutions to diverse applications. The course also highlighted the improvements offered by Ada 9X in writing asynchronous tasks and in supporting object-oriented design. The instructor included coding examples to help make Ada 9X features seem more real to the students.

The tests stressed the Ada language features that bore directly on the needs of safety-critical software engineering. The midterm and final exams, which expresses this emphasis, are included in Attachments E and F, respectively.

To obtain a larger enrollment, IIT offered this course over a closed-circuit TV network (IITV). This allowed IIT to tape the course in VHS format. IIT is delivering this tape as part of the Final Report material.

RESULTS AND CONCLUSIONS

At the end of the Final Exam, the instructor asked if anyone felt his or her goals in taking the course were NOT met. Nobody responded, suggesting that the goals had been met reasonably well. Several students volunteered after class that they had really enjoyed this course__despite the work involved in covering so much material so quickly.

All indications are that the course did give the students a good sense of the benefits of using Ada in safety-critical applications. The payoff will come when they join industry software groups and spread the Ada message to their colleagues.

FOLLOW-UP PLANS

1. IIT has scheduled this course to be given again in the Spring semester. IIT will make another video tape, with the idea that the second offering of a new course usually goes more smoothly than the first.

2. IIT is offering other courses using Ada, such as an Operating Systems course. This is possible now that a course that actually teaches the language is available in the curriculum. Ada is the teaching language of choice for these courses because of its power and expressiveness.

3. IIT plans to disseminate the improved video tape of this course to instructors at other colleges and universities who are in the process of developing a similar course.

4. IIT plans to purchase several copies of Ada compilers and Ada tutorials that can be run by individual students on PCs. This will allow students in courses that use Ada to learn or relearn it at their own pace.

REFERENCES

1. John Barnes, "Introducing Ada 9X," *ACM Ada Letters,* Nov/Dec 1993

2. Levenson & Turner, "An Investigation of Therac-25 Accidents," *Computer,* July 1993.

3. F. Houston, "What Do the Simple Folk Do?: Software Safety in the Cottage Industry," *IEEE Computers in Medicine Conf.*, 1985.

ATTACHMENT A

ADA CLASS ANNOUNCEMENT

CS 495 SAFETY-CRITICAL SOFTWARE ENGINEERING WITH ADA

Who should enroll? Software students and practitioners with at least one year of programming experience.

In this course you will learn to:

* Use Ada, the language being adopted world-wide to implement the most difficult software systems.

* Use Ada programming features that help NASA, the FAA and the military develop software for systems that can't afford surprise behavior.

* Use Ada programming features that help one of the world's largest telephone companies to develop systems that work better and cost less.

* Avoid the three technical problems facing new Ada users.

* Use Ada 9X with Object Oriented Analysis and Design methods.

* Double your personal programming productivity__and then double it again with more practice.

* Understand how to change your approach when you are a member of a very large software engineering team.

When: IIT Summer Session, Fridays, 3:10 to 6:50 PM.

Where: Rice Campus. Also on TV for other locations.

Credit: Two credit hours. (Discuss with contacts listed below.)

Text: Grady Booch, Software Engineering With Ada

Instr: Fred Francl managed real-time, mission-critical software engineering development for over fifteen years. He led several government studies on the effectiveness of Ada for this type of software development. He served for three years as a Distinguished Reviewer for an Ada Joint Program Office team. Mr. Francl led the Real-Time Session of a national Ada conference. He currently consults with the Federal Aviation Administration on Ada issues that arise in the modernization of the U.S. Enroute Navigation System. He chairs the Chicago Chapter of the ACM Special Interest Group on Ada (SIGAda).

ATTACHMENT A

Contacts: Dr. Tzilla Elrad (312)567-5142 CSELRAD@minna.acc.iit.edu
Mr. Fred Francl (708)627-8098 ffrancl@ajpo.sei.cmu.edu

ATTACHMENT A

ATTACHMENT B

ADA CLASS SURVEY RESULTS

Student #	Class Learning Goals
1	Ada
2	Ada
3	How Ada fits applications, how it supports OO
4	Real-time OO and Software Engineering principles
5	Ada
6	Ada and Software Engineering principles
7	Applying Ada to real-world problems
8	Ada knowledge
9	Ada knowledge
10	Relearn Ada (learned from manual 10 years ago)
11	Ada

ATTACHMENT B

ATTACHMENT C

TEXT CHAPTER WEIGHTING

Ch.	Weight	Rationale
1	A	Includes lecture material not in book
2	A	Includes lecture material not in book
3	F	
4	A	Especially 6-step Booch OO method
5	A	
6	C	
7	D	
8	B	
9	D	
10	C	
11	C	
12	D	
13	A	
14	C	Skip 14.2, 14.3
15	D	
16	B	Be able to write task specs, to read task bodies
17	B	Skip 17.2, 17.3
18	D	
19	C	Skip 19.3
20	B	
21	D	
22	F	To be covered after Final Exam
23	F	To be covered after Final Exam
24	F	To be covered after Final Exam

A = Almost sure to be important in the final exam

.

.

F = Definitely not exam material. For familiarization only.

ATTACHMENT C

ATTACHMENT D

SAMPLE VIEW GRAPHS

ATTACHMENT E

MID-TERM EXAM

ADMINISTRIVIA

Please take this exam with a closed textbook and with a one hour time limit. When complete, give to the proctor and leave the classroom. Please return one hour after the test start time.

Record your answers on a sheet of your own paper with your name, "CS495 Mid-Term Exam" and today's date at the top of the page. If multiple pages are used, put your name and the page number at the top of each sheet.

PROBLEM TO SOLVE

Use the Booch Object-Oriented (OO) Development Method to specify an Ada software simulation of an automobile cruise control system. Assume it is to be run by a user__called the *driver*__who accesses each of the driver controls through the terminal and keyboard.

PROBLEM APPROACH TO USE

The first step in the Booch OO Development Method, Define the Problem, is provided on the following page. The final step, Implement Each Object, is not required. Doing the final step__or some parts of it__earns extra credit, but only if done well.

For each step in the Booch OO Development Method, enter and explain the results of your analysis on your test paper. For example, if a Booch OO Development Method step requires you to identify certain kinds of items, list the items on your test paper and explain your reason for choosing them.

Any code or pseudo-code produced should follow the Ada syntax rules as much as possible, but no points will be deducted for syntax errors. Please use comments to assure that your Ada code will be understandable even if the syntax is wrong.

Your solution will be graded on:

METHOD (How well the Booch OO Development Method is followed)
COMPLETENESS (Including any exceptions needed for safety)
CORRECTNESS (With respect to the statement of the problem)
SIMPLICITY (No complexity not required by the problem)
READABILITY (Of Ada code or Ada pseudo-code produced)

STEP 1: DEFINE THE PROBLEM

A real automobile cruise control system maintains the speed set by the operator (*driver*) by pressing the accelerator when going up hills and releasing it when going down hills. This cruise control system shall be an *abstraction* or *simulation* of a real one.

The compiler shall supply the package *Clock*, which may be imported:

```
package Clock is
   type Time_Type is private;
   function Current_Time
         return Time_Type;  --current clock time to the nearest msec
   function Add_1_Second
      (Current_Time: in Time_Type)
         return Time_Type;        --adds one second to Current_Time
   function Timer_Expired
      (Current_Time: in Time_Type
      Timer_Set_Time: in Time_Type)
         return Boolean;    --True if Current_Time > Timer_Set_Time
private
   type Time_Type is range 0..Long_Integer'Last;
end Clock;
```

The driver shall control the simulation with the following inputs:

1. *ON* shall display the current speed, randomly change speed each second by a small amount, and respond to all keys described below if the current state is *OFF*. Otherwise it shall do nothing.

2. *OFF* shall stop displaying the current speed, ignore all keys except the *ON* key, and erase any set speeds from memory if the current state is *ON*. Otherwise it shall do nothing.

3. *ACCELERATE* shall increase speed at a constant rate each second while its key is depressed. When its key is released the current speed shall be stored in memory as the *set speed*. This speed shall then remain constant.

4. *COAST* shall decrease speed at a constant rate each second while its key is depressed. When its key is released the current speed shall be stored in memory as the *set speed*. This speed shall then remain constant.

5. *BRAKE* shall decrease speed at a constant rate ten times that of the *COAST* key each second while its key is depressed. When its key is released the current speed shall be randomly changed by a small amount each second. The *set speed* (if *ACCELERATE* or *COAST* had previously stored it) shall remain in memory.

6. *RESUME* shall accelerate at the same rate as the *ACCELERATE* key or decelerate at the same rate as the *COAST* key until the *set speed* is reached. This speed shall then remain constant.

ATTACHMENT F

FINAL EXAM

Name:________________________

(USE BACK OF PAPER IF YOU NEED MORE ROOM TO ANSWER ANY QUESTION)

1. Assume each member of our class is to write a 100 page paper. If there are no misspellings, grammatical mistakes, or factual errors in <u>any</u> of the papers, then assume each member of the class gets $100,000. But if there is even one error in <u>any</u> of the papers, assume each member of the class will be killed.

   a. What things would you suggest doing or what things would you suggest obtaining to improve the chances for success?

   b. What things would you suggest doing or obtaining if each person had to write an error-free 10,000 lines of code portion of a software program instead of a 100 page paper?

   c. Assume there is a time limit of two years to finish the class software project. Further assume that the class has the usual distribution of software engineering productivity, and that the best in the class is ten times as productive as the worst in the class. What would you suggest doing to speed up the project to minimize the chances of being late without introducing fatal errors__literally__into the code?

2. Ada has been called a "large" language. Compared to Pascal it has many more built-in features, supporting such things as data abstraction, concurrent processing and machine-level operations (such as being able to load a register at a specific address with a specific hexadecimal value.)

   a. When would it be an advantage to use a large language like Ada with these capabilities?

   b. When would you be better off with a smaller language like Pascal?

3. Assume you work for a company that has standardized on the use of a single language for all its projects. This happens because companies often expect this standardization will make it easier to transfer engineers among projects and to reuse code.

   a. What are the added advantages if that single language is Ada?

   b. It is possible to write Ada-like code in another language, but the compiler will not enforce things like visibility rules. What are the advantages of designing as if you could use Ada and then enforcing needed rules__such as object scope and visibility__yourself?

4. Assume you work for a company that has standardized on a single design method. This happens because companies often expect this standardization will reduce expenditures for method support tools and/or method training courses.

   a. What are the added advantages if that single design method is Object-Oriented Development?

   b. What is the fastest-growing development method today?

5. A "hacker" can be defined as someone who designs software as quickly as possible, who loves to include optimizations, and who thinks a program is good enough if it "works" for his tests.

   a. What is your definition of a software engineer?

   b. When does a good software engineer add optimizations to the code?

   c. In a large, complex software project, we have learned in this course that *Quality if free. It is lack of quality that costs money.* Explain why this is true.

6. **The first step in the Booch Object-Oriented (OO) Development method is to define the problem.**

   a. **How do you use the problem definition to identify the objects?**

   b. **How do you use the problem definition to identify the operations on each object?**

   c. **How do you represent operations using Ada?**

   d. **What kind of a diagram is suitable to establish the *visibility of each object*?**

   e. **Which Ada language construct is especially suited to establishing the interface for each object?**

   f. **Which Ada language construct is especially suited to implementing each object?**

7. **Ada is better than many traditional languages that were designed to support only procedural abstractions (such as could be gotten from a traditional flow chart.) Ada is designed to support data abstractions as well as procedural abstractions.**

   a. **What Ada language construct do you use to express a data abstraction?**

   b. **If only certain operations make sense to be used with this abstraction, how do you show this in Ada?**

   c. **If the data abstraction has internal states (for example, is the German Shepherd object asleep or awake?) that must be considered to solve the problem, how do you show this in Ada?**

   d. What is an *encapsulated* type?

   e. How is an encapsulated type shown in Ada?

   f. What is a type *attribute*?

   g. **If a data abstraction occurs for a number of different types of object (such as a queue that can accept integers, real numbers, character strings, etc.), what Ada language construct do you use to avoid rewriting the abstraction for each type of object?**

8. Older, traditional languages need to use data dictionaries and set/use tables (which show everywhere a variable value is set or used) to keep track of variables.

   a. What features in Ada__if used properly__make data dictionaries unnecessary?

   b. Certain variables are only used to indicate that abnormal processing must be done, or that an error condition has arisen. What special type does Ada assign to these variables?

9. Tasks are used to express concurrent action in Ada. They operate synchronously using a mechanism called the *rendezvous.*

   a. What information about a task can you find in the task specification?

   b. How does an operating system decide which task should run?

   c. How do you show *asynchronous* concurrent action (for example, a mailbox) using Ada tasks?